MODELING USER-AFFECTED SOFTWARE PROPERTIES FOR OPEN SOURCE SOFTWARE SUPPLY CHAINS

A Dissertation Presented for the
Doctor of Philosophy
Degree
The University of Tennessee, Knoxville

Tapajit Dey

August 2020

ACKNOWLEDGMENTS

I would like to start by thanking Dr. Audris Mockus, my advisor, for encouraging me to ask important open research questions that make strong contributions to the open source software community and for helping me throughout my PhD on academic as well as non-academic matters. I also thank my PhD committee for their time, support and guidance: Dr. Russell Zaretzski, Dr. Austin Henley, and Dr. Jian Huang. A special thanks to Dr. Randy Bradley for introducing me to the concept of Information System Success Model, which has served as the conceptual basis for my research.

I would also like to thank my former labmate, Sadika, for her support and encouragement, and my colleagues, Yuxing and Sara, for their assistance and friendship, and a special thanks to our very helpful friend, Eduardo. I'd also like to thank other fellow students I worked with at various points of time in the department and beyond.

I also am eternally grateful to my parents and other relatives for their continued love and support, and a special thanks to my wife, Soumee, for being there with me in the difficult times brought upon by the pandemic.

Finally, I'd like to thank the Open Source Software development community for their amazing work in making the lives of everyone on the planet better.

Note: The work has been partially supported by the following NSF awards: CNS-1925615, IIS-1633437, and IIS-1901102. All opinions, findings, conclusions, or recommendations expressed in this document are those of the author and do not necessarily reflect the views of the sponsoring agency.

ABSTRACT

Background: Open Source Software development community relies heavily on users of the software and contributors outside of the core developers to produce top-quality software and provide long-term support. However, the relationship between a software and its contributors in terms of exactly how they are related through dependencies and how the users of a software affect many of its properties are not very well understood.

Aim: My research covers a number of aspects related to answering the overarching question of modeling the software properties affected by users and the supply chain structure of software ecosystems, viz. 1) Understanding how software usage affect its perceived quality; 2) Estimating the effects of indirect usage (e.g. dependent packages) on software popularity; 3) Investigating the patch submission and issue creation patterns of external contributors; 4) Examining how the patch acceptance probability is related to the contributors' characteristics. 5) A related topic, the identification of bots that commit code, aimed at improving the accuracy of these and other similar studies was also investigated.

Methodology: Most of the Research Questions are addressed by studying the NPM ecosystem, with data from various sources like the World of Code, GHTorrent, and the GiHub API. Different supervised and unsupervised machine learning models, including Regression, Random Forest, Bayesian Networks, and clustering, were used to answer appropriate questions.

Results: 1) Software usage affects its perceived quality even after accounting for code complexity measures. 2) The number of dependents and dependencies of a software were observed to be able to predict the change in its popularity with good accuracy. 3) Users interact (contribute issues or patches) primarily with their direct dependencies, and rarely with transitive dependencies. 4) A user's earlier interaction with the repository to which they are contributing a patch, and their familiarity with related topics were important predictors impacting the chance of a pull request getting accepted. 5) Developed **BIMAN**, a systematic methodology for identifying bots.

Conclusion: Different aspects of how users and their characteristics affect different software properties were analyzed, which should lead to a better understanding of the complex interaction between software developers and users/ contributors.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

OSS	Open Source Software
FLOSS	Free/Libre and Open Source Software
SSC	Software Supply Chain
WoC	World of Code
IS	Information System
PR	Pull Request

CHAPTER 1
INTRODUCTION

1.1 Overview

According to the IEEE Standard Glossary of Software Engineering Terminology, [1], Software Engineering is defined as *“The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.”* Functionally, the goal of this discipline is development and maintenance of software, and a number of sub-disciplines focus on specific aspects of the task, e.g. specification and validation of the requirements of a software; defining the architecture, components, and other characteristics; construction of the software by a combination of programming, debugging, and testing etc. On the other hand, the goal of software engineering research is understanding the nature of software and its usage. Empirical software engineering, which is a sub-field of software engineering research, is concerned with empirical validation of various software engineering theories and assumptions, the results of which can give us a deeper understanding of the interrelationship among various factors that influence different software properties, and that knowledge can be used to create better software.

The contribution of this thesis concerns integrating the conceptual frameworks of Information System and Software Supply Chain, identification of software properties related to user interaction with different software modules that are part of larger software ecosystems,[1], and modeling the effects of such interaction and the interdependence among the modules in the ecosystem on those properties. This subject of this research is primarily Open Source Software, where the importance of user interaction with the software is particularly prominent, since the source code is publicly available and can be modified and reused with limited restrictions, and contributions from developers outside of the core team are encouraged. With over 100M projects, OSS has taken a critical role in the digital

[1]“A collection of software projects, which are developed and co-evolve in the same environment, and can share code, depend on one another, share developers between themselves, reuse the same code, and can be built on similar technologies.” [2]

infrastructure of modern society [3]. This serves as a motivation and provides urgency to to our research objective of understanding the nature of this dependency structure in order to understand OSS development at an ecosystem level.

1.1.1 Software as an Information System: The Importance of Users

Modern software is more complex than just a piece of code, it is in fact more akin to an Information System (IS), since a it is a sociotechnical system that essentially collects, processes, and stores information [4]. From that perspective, the overall utility of a software should depend on factors similar to the ones that affect the success of an Information System. In their paper, Delone and McLean [5] proposed the **Information System Success Model**, which lists different factors influencing the success of an Information System and the interrelationship among those factors, as can be seen in Figure 1-1.

This concept can be applied to the domain of software engineering as well, and it can be useful for understanding which factors influence the success of a software as a product. Among such factors, the "Information Quality" is analogous to the features of a software, the "System Quality" aspect can be thought of as the quality of the code of the software, and the "Service Quality" is similar to how the team maintaining the software handles issues, patches, and feature requests. These are the aspects of the software that can be thought of as "internal" to the software. However, as seen from Figure 1-1, these factors do not have any direct influence on the success of the software. The success of a software, or the "Net System Benefit" is affected by how many users use or are willing to use the software, and how satisfied they are with it. As we can see from Figure 1-1, there is, in fact, a feedback loop between these aspects of the system. The success model indicates that by only focusing on the internal aspects of a software, we would overlook valuable insights which would be crucial for determining the overall success of the software.

The importance of looking beyond the "internal" aspects is even more important for Open Source Software (OSS), which relies heavily on regular contributions from contributors outside of the core team of developers to produce and maintain the quality of the software. Therefore, understanding how the users of a Open Source Software, who might potentially contribute to the software or even become part of the core team at some point, interact with the software and affect its properties is crucial for understanding and ensuring the success of individual software projects as well as the Open Source Software development paradigm as a whole. When referring to the "users" of a software in this

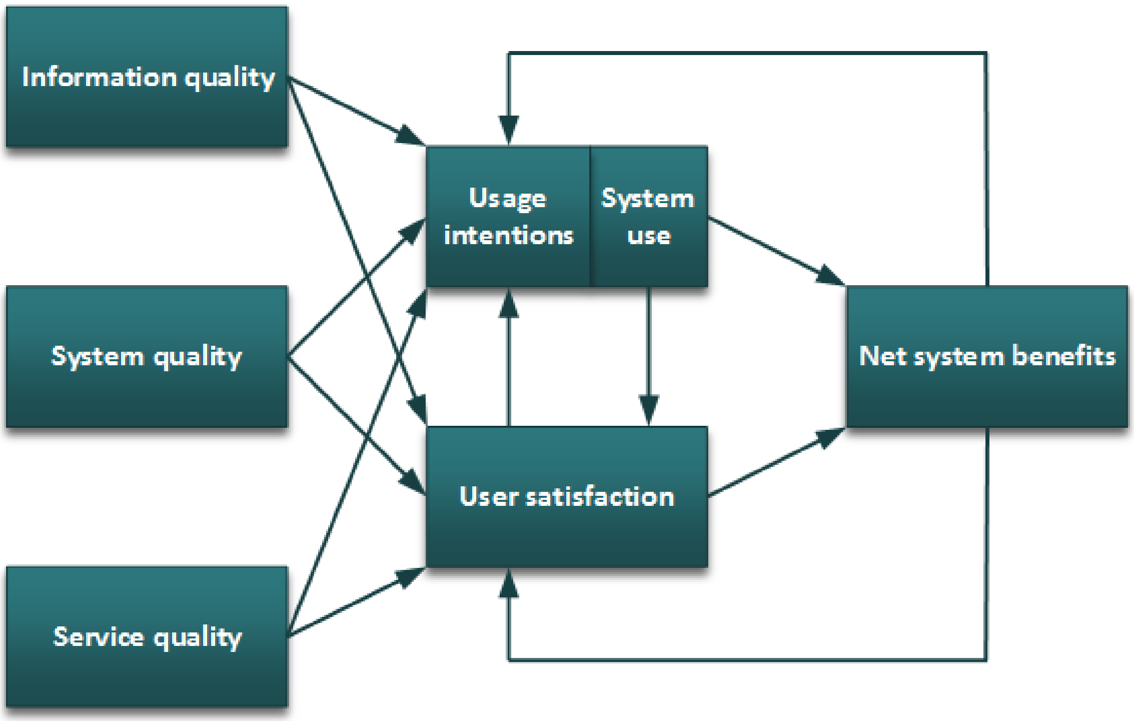


Figure 1-1. Information Systems Success Model [5]

dissertation , we are essentially referring to these user-developers, who use various OSS projects, and do, or might, contribute to the projects at some point.

The Information System Success Model (Figure 1-1) suggests that user interaction with a software doesn't affect its internal properties, however, that is not completely true for OSS systems. The structure of various social-coding platforms (e.g. GitHub, GitLab, BitBucket etc.), which are commonly used for the development and maintenance of OSS, allows a user to communicate with the developers of a software by creating issues, and contribute code to the software projects by creating Pull Requests. Other ways of communicating with the developers include joining the project mailing lists or getting a hold of individual developers through other means (personal email/ meeting them in a conference etc.). Thus, the users can actually contribute to the internal code of the software, and can have some influence on the core team's decisions, e.g. which features might be added/ removed from the software, as well.

Figure 1-2 shows the software properties analogous to those of an Information System, as shown in Figure 1-1. However, the total number of different software properties is huge, and Figure 1-1 doesn't show all of them. The purpose of creating this analogue is

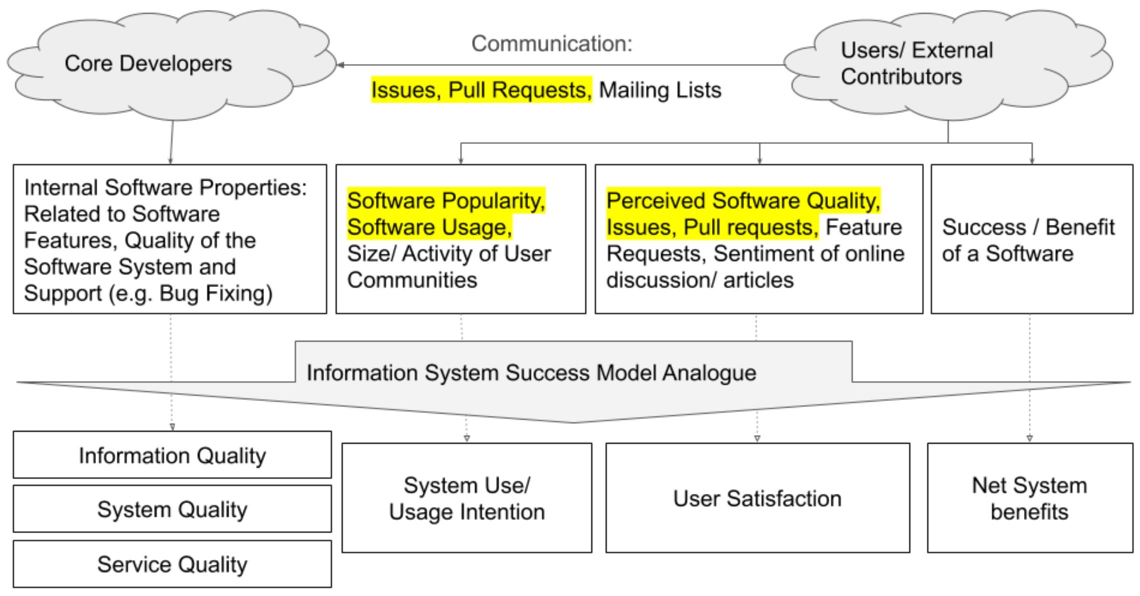

Core Developers
Communication:
Issues, Pull Requests, Mailing Lists
Users/ External Contributors
Internal Software Properties: Related to Software Features, Quality of the Software System and Support (e.g. Bug Fixing)
Software Popularity, Software Usage, Size/ Activity of User Communities
Perceived Software Quality, Issues, Pull requests, Feature Requests, Sentiment of online discussion/ articles
Success / Benefit of a Software
Information System Success Model Analogue
Information Quality
System Quality
Service Quality
System Use/ Usage Intention
User Satisfaction
Net System benefits

Figure 1-2. Software Properties Analogous to Information System Properties. The highlighted Properties were investigated in this dissertation .

to highlight the properties studied in this research, and how they are related to different properties of an Information System, as well as showing a few software properties related to each IS aspect that were not covered by this research, in order to place this research in a broader perspective. As shown by Figure 1-2, the "internal"properties of software, e.g. its features, the software development and maintenance process etc. are related to the aspects of "Information Quality", "System Quality", and "Service Quality", and are primarily worked on by the core developers of the software. These properties were not studied by this research, since the external contributors/ users of a software do not have any direct effect on them. The "System Use/ Usage Intention" aspect of an IS are analogous to the software popularity, software usage, and the size and activity of user communities. Example of software properties analogous to the "User Satisfaction" aspect are the perceived quality of a software (the no. of software faults experienced by users), the amount of issues, PRs, feature requests created by the users, and the overall sentiment of online discussions/ blogs/ articles related to the software. The concept of "Net System Benefits" is rather illusive in comparison, so we refer to the overall success of the software as an analogue to this aspect.

The specific properties discussed in this dissertation are related to software usage, popularity, its quality from the users' perspective (the number of software faults experience by the users), and the communication between the users of a software and the core developers through issue and PR submission. The rationale behind choosing these properties and the exact research goals are discussed in section 1.2.

1.1.1.1 Previous Studies on Software Users

The importance of users in open source software development is well recognized in general, e.g. it was highlighted in previous studies that the users of a software help test the software by using and interacting with it [6]; they help increase popularity of a software and/or the domain as a whole by *"Spreading the Word"* [6]; they may also provide feedback, suggestions, and ideas about functionality [7, 8] that eventually help drive the direction of the development process; additionally, the users report and, in many cases, fix bugs [9] as well. The users also motivate developers and give credibility to a project [6] and some users support developers directly by donating [6] to the projects.

However, while previous studies on the users of open source software focused on identifying the nature of their interaction with the software [6, 7, 8, 9], investigation of user communities (e.g. how they are formed [10], how they support each other [11], the tools they use for communication [12] etc.), and focused on other aspects such as modeling user expectations [13], mimicking user interaction patterns for the purpose of testing [14], and predicting user behaviour for creating Web UI or adaptive systems [15, 16, 17], few studies focused on identifying how the user interaction with a software affects its properties, which is a crucial piece of information required to understand the complex dynamics that dictates the ultimate usefulness of a software. As of now, over 42 million developers work on more than 123 million different OSS projects, [2] giving rise to complex interaction patterns among themselves. Deciphering the effect of such interaction is a necessary step towards developing better measures for different software properties and an improved understanding of how to develop better software.

1.1.2 Dependency among Software Modules: Software Supply Chain

While the concept of Software Engineering has been around for over 50 years, [18] the process of software development has gone through some drastic changes over the years. At present, most software projects are not monolithic, instead focusing on using functionalities available in existing modules and incorporating incremental changes, which are made available to the users of the software in the form of rapid releases. While this

[2]https://bitbucket.org/swsc/overview/src/master/

process entails many benefits, e.g. faster turnaround time, avoidance of code duplication etc., it also leads to the creation of an interconnected network of dependencies among different software modules.

The interconnected dependency network between software modules closely resemble the structure of a Supply Chain [19, 20], and it is possible to address many of the associated problems by leveraging the existing knowledge in traditional supply chain management. In fact, the supply chain view can be used to describe not only the interdependent software modules, but the relationship between developers working on different projects and the interaction between the developers and the code in a project as well.

Traditional Supply Chain Management (SCM) encompasses a system of organizations, people, activities, information, and resources involved in moving a product or service from supplier to consumer [19]. It has been extremely successful in helping businesses manage risks that arise from the distributed nature of production by the integration of supply chain activities through improved supply chain relationships to achieve a competitive advantage. The ultimate goal of SCM is to ensure that merchandise is punctually produced and distributed in the correct quantities and to the correct locations in order to minimize system-wide costs, all the while satisfying the service level requirements. However, unlike a physical supply chain, software as the source code neither requires transportation nor does it accumulate production costs. Software Supply Chain (SSC) can, therefore, be defined as the collection of developers in software projects producing new versions of the source code [20].

While it is possible to create different types of SSCs [20], e.g. Code Reuse Networks, Knowledge Flow Networks etc., for this research, we focused only on the Software Dependency Networks that depict the interdependence between different software modules. Knowing the interdependence between software modules lets us conceptualize a dependency networks between the developers, contributors, and users of different projects, and can, therefore, be enormously helpful in understanding precisely how users interact with a project and how they might depend on a project.

1.1.2.1 Previous Studies Discussing Software Supply Chain

The first notable use of the term “Software Supply Chain” was in 1995 [21], and the concept has since been used for addressing economic and management issues in software engineering [22], for facilitating the Software Factory development environment introduced by Microsoft [23] and elsewhere. The primary use of software supply chain has been for identifying and managing risks related to the software development process [24, 25]. The “merchandise” in SSC is the activity of millions of developers and organizations to innovate, to improve quality, and to adapt software to constantly changing environment. Without this activity OSS would stop functioning and lead to dire consequences to all who rely on it. These actions may be recorded in the form of issue reports or source code changes or take the form of knowledge and information acquired or exchanged by the developers or embedded in the software they produce. This supply chain analogy provides us with key concepts useful for abstracting the complex software dependency networks involving nodes that represent developers, changes, projects, and files, and also familiarizes us with notions of “transperency” and “visibility” [20], which are important for assessing the strengths and weaknesses of different software ecosystems.

1.2 Research Goals

While there have been a plethora of studies looking at different aspects of a software and its properties, few studies investigated how the users of a software affect its properties. However, according to the conceptual framework of Information System Success Model, [5] as described earlier, the ultimate success of a software is heavily dependent on its users and their effect on the software. Moreover, most of the studies investigating various theories and assumptions about software engineering were conducted on individual projects, or few isolated projects, which limits the generalizability of those studies for the interconnected software modules and software ecosystems. Adopting a perspective of software supply chain can, therefore, lead to new insights about the contemporary software landscape and produce useful insights applicable to the software ecosystems.

In this research, which relates to the field of empirical software engineering, we focused on the software properties that are affected by the users’ interaction with a software as well as by the interdependencies among various software modules (modeled as a supply chain) at an ecosystem level. The “internal” properties of a software, e.g. its features, the development process, strategies, and maintenance efforts do not fall in the desired category, since they are not directly affected by the users. These properties are related to the aspects of Information Quality, System Quality, and Service Quality, as shown in Figure 1-1. On the other hand, the properties of a software like its usage and popularity, its quality from the perspective of users, and various properties of the user communities that dictate the users’ experience with the software are indeed affected by its users. These properties are related to the aspects of System use, User Satisfaction, and Net System Benefits of an Information System, as shown in Figure 1-1.

It is easy to see how the popularity of a software would be influenced by the supply chain structure of a software ecosystem, since all dependencies of a software module are typically downloaded along with it, influencing their usage. The amount of usage of a software, however, also affects its perceived quality [26, 27, 28, 29], therefore it is also affected by the supply chain structure indirectly. Another aspect without which any study on the users' effect on OSS would be incomplete is how they communicate with the core developers of a project, because the users can, in turn, affect a software by submitting their feedback as well as making code contributions, especially in the OSS development scenario. The primary modes of communication between the users/ external contributors of a project and the core developers are through submission of issues and PRs, and communication via mailing lists. We, however, focused only on issues and PRs, since they are transparent and trackable through the social coding platforms. While there have been a good number of studies about issues and PRs, we focused on a specific aspect in this research that wasn't covered in previous studies: How the issue creators are connected to the projects to which they create issues through the software dependency supply chain, and if the developers' connection to a project and their overall experience in OSS development have any impact on the probability of their PRs being accepted.

All of the research topics mentioned above assume that the users in question are humans, because automated programs and applications, or "bots", might have very different characteristics and motivations and can interact with a software in very different ways. Therefore, for preserving the integrity of the research results, it is necessary to distinguish between human users and bots. However, detecting whether a user in a social-coding platform is not always straightforward. So, we undertook a project for developing a systematic method for detecting bots that commit code. This method can be expanded in future to identify bots that create issues or PRs as well. Since the study is an add-on to the main theme of the research, it is presented in Appendix A.

1.2.1 Specific Research Topics and Scope of the Problems

In order to address the research goal of modeling different software properties that are affected by users as well as the supply chain structure of the software ecosystems, we investigated the software properties of interest, as shown in Figure 1-2. The specific research topics explored by my research are listed below, along with the scope of each problem.

1.2.1.1 Perceived Quality of a Software

RT1: Investigating if the perceived quality of a software depend on its usage even after accounting for the code complexity measures, and, if we can design a easy-to-use and usage independent measure for software quality.

Although software quality is a multifaceted measure, for our research, we focused on how it can be used to measure the quality of experience of the users of a software, addressing the aspect of “User Satisfaction”. How the users feel about a software is significantly affected by how many problems, e.g. crashes, bugs etc. they face while they are using the software. Therefore, the the number of software failures experienced by the users of a software is used as the measure for the perceived quality of the software. In order to find out the effect of user interaction on this measure, we investigated a proprietary software from the telecommunications domain as well as a number of popular packages from the NPM ecosystem. The number of new users and the number of downloads were used as measures for user interaction for the two scenarios respectively.

The reason for choosing the proprietary software was that it was a rare instance where we had access to detailed user-level data for an industrial software, so we decided to test our hypothesis that the perceived quality of a software depends on usage using this dataset. The NPM packages were used to test the external validity of the concept, since we used proxy measures for usage as well as for software failure. The numbers of downloads and issues, respectively, were used as measures for the two properties.

1.2.1.2 Effects of Indirect Usage on Software Popularity

RT2: Modeling the effect of indirect usage (e.g. dependent packages) of a software on its popularity.

We focused on the popular packages in the NPM ecosystem for measuring the influence of indirect usage on software popularity. The numbers of dependencies and dependents of an NPM package were used as a measure for its indirect usage, and the popularity was measured by the number of downloads. The reason for focusing on the NPM ecosystem was due to the fact that the number of downloads for individual packages, which is a good measure of popularity, was readily available for this ecosystem.

1.2.1.3 Relationship between the Issue Creators and Software Packages through the Software Supply Chain

RT3: Exploring if the issue contributors of OSS projects depend on the project through dependency supply chain, and if the contributors who aren't dependent on the package through the supply chain have a different characteristics than the contributors who are.

The focus of this work was once again on the NPM ecosystem. We collected all the issues created against a number of popular NPM packages using the GitHub Rest API, discovered all of the projects the issue creator contributed to using data from GHTorrent[3] and the World of Code (WoC) dataset [30], and searched for any dependency between the projects contributed to by the issue creator and the packages to which the issues were created. The primary reason behind focusing on the NPM ecosystem in this case was its size and popularity among the contemporary software developers, which made it a good target for studying.

1.2.1.4 Effects of Pull Request (PR) Creators' Characteristics on Pull Request Acceptance

RT4: Investigating the effects of the characteristics of individual patch contributors, specifically, their experience and social proximity to the repositories to which they submit PRs, on the chances of their pull requests getting accepted.

Once again, we focused on a number of popular NPM packages for addressing this question. Data on all of the Pull Requests created for those packages was collected using the GitHub Rest API, and further data on the PR creators was collected using the WoC dataset [30]. The reason for focusing on the NPM ecosystem in this scenario is similar to the last once, i.e. the size and popularity of the NPM ecosystem.

1.2.1.5 Detection of Bots that Commit Code

Appendix - RT5: Designing a systematic method for identifying bots that commit code to various social-coding platforms.

To address this research goal, which would be useful for cleaning the datasets used in empirical studies similar to the ones we conducted, and improve the accuracy of the studies, we collected data about all developers who have committed code in the projects that use the git version control system, and used a number of heuristics based on their characteristics

[3]https://ghtorrent.org/

for determining if they are bots. The methodology for detecting bots was developed using detailed data collected from the WoC dataset [30]. The reason for focusing on the git version control system was that it is the single most popular version version control system at present. The result of this project is presented in Appendix A, since the topic is related to the theme of the research, but not directly a part of the theme.

1.3 Contributions

We investigated a number of facets of the complex interaction between the users of a software and its various properties and try to model the effects of such interaction. In particular, this research makes the following contributions:

1. The results of our research show that software usage affects the perceived quality of a software even after accounting for other code complexity measures, and a measure of quality, defined by the number of software faults per user (or a similar usage metric), that is independent of software usage and can be used to objectively compare the qualities of different software releases or modules. The result of this research was published in a conference paper [26] and a journal article [27].

2. The amount of indirect usage was found to have a very strong effect on the popularity of a software: it could explain between 80%-100% of the variance in popularity of a software, and was even able to predict whether the popularity will increase or decrease with an AUC-ROC value of 0.73. This shows us the strong dependence between the popularity of a software and the interconnected nature of the ecosystem, which is a useful knowledge for the researchers in the field as well as for the software developers who can leverage this interdependence for increasing the popularity of their software. Some of the results of this research was published in a conference article [31].

3. A large number of user-contributors who create issues and PRs to a project was found to have contributed to some project that depends on it directly, while contributions to projects on which their projects have a transitive dependency was found to be very rare for the NPM ecosystem. This finding reveals a lack of visibility and transparency in the ecosystem, which should be remedied for increasing the long-term sustainability of the ecosystem. The results of this research was published in a conference paper [32].

4. The overall experience and the track record of PR creators, as well as characteristics of the repository to which they are contributing have a strong influence on PR acceptance probability. The result of this research is a model that predicts PR acceptance with an AUC-ROC value of 0.94, and it can be converted into a tool that can help the PR integrators in prioritizing PR reviews. This research also found the exact nature of the functional relationships between the predictors and the probability of PR acceptance, which can help the developers to know and prioritize the aspects they should focus on to get their contributions accepted (for the PR creators), or to gauge the quality of a submitted PR (for the integrators), or trying to decide which signals to make available to the parties involved (for tool designers). The results of this study was accepted in the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2020 [33].

5. **BIMAN** (Bot Identification using commit Messages, commit Association, and committer Name), a systematic process of detecting bots was proposed that can detect if a given code committer is a bot with an AUC-ROC value of 0.9. The method was used to detect 461 bots with over 1000 commits and a dataset containing 13,762,430 commits made by these bots was created and shared publicly. These bots were further characterized for identifying the languages they are active in, their activity patterns, and the types of files they modify. This research was conducted in collaboration with multiple researchers, but the majority (over 80%) of the work was done by me. The result of this research was accepted in the Mining Software Repositories (MSR) conference, 2020 [34].

1.4 Dissertation Organization

This dissertation is comprised of seven chapters. Chapters 2-5 discuss the four primary research topics proposed earlier, in order, in further detail, including specific questions that explore the particular topic in greater detail, previous works related to the topic, methodologies used for addressing the specific problems, and the results corresponding to the specific questions. Chapter 6 concludes the dissertation by reiterating over the research questions and their answers, as well as the summary of contributions made in this dissertation, and also proposes some ideas about the future work. The results of the bot detection project is presented as an appendix to the main dissertation in Appendix A.

CHAPTER 2

MODELING THE RELATIONSHIP BETWEEN SOFTWARE USAGE AND PERCEIVED SOFTWARE QUALITY

2.1 Overview

This chapter addresses the topic of exploring the relationship between software usage and software quality, and aims to propose a software quality metric independent of software usage. Thereby, it tries to answer the first research question posed in subsection 1.2.1 of Chapter 1: "Does the perceived quality of a software depend on its usage even after accounting for the code complexity measures, and how can we design a easy-to-use and usage independent measure for software quality?" Most of the research about this topic was published in a paper [26] in the PROMISE conference in 2018, and an article [27] in Empirical Software Engineering journal.

2.2 Background

Improving the quality of a software has always been a key goal for software developers. However, the term "Software Quality" means different things in different contexts, and in our context of investigating the effects of user interaction on it, we refer to the quality of a software as perceived by its users. The level of user satisfaction (which is a key factor influencing the success of a software Figure 1-1) is heavily influenced by how many problems, e.g. bugs, crashes, or similar issues, a user encounters while using a software. Therefore, for the purpose of this research, we define the perceived quality of a software as the number of software faults encountered by its users. In light of this definition, since the users only use the post-release versions of a software, any pre-release faults or bugs of the software are not considered while measuring its perceived quality. An intuitive measure of the software quality, therefore, would be the total number of software failures, and the qualities of two different software versions or different software packages can be compared by simply comparing the number of bugs/ crashes/ other measures of software failure.

This measure of software quality, however, can be directly influenced by its usage, e.g. in an extreme case, a software with no users would have no reported bugs. This interdependence of software usage volume and crashes experienced is typically not considered while measuring its quality in industry or in empirical studies (although few studies do note that [35, 36]), which can lead to grossly misleading conclusions, and would misguide quality improvement efforts (avoid quality improvements for releases/ software with low usage) and/or misguided developer performance metrics (reward developers of low-usage products). This analogy is applicable for software crashes, defects (bugs), and, by extension, for issues raised against a software, since software crashes are manifestations of underlying defects and [37, 29] observed that the number of discovered software defects increases with the number of users, though the relationship between crashes and defects is not very well understood [36].

The goal of this project was twofold, as described by the research question (RQ1) we're trying to answer: (1) Investigating if the number of software faults, which is a measure of the perceived quality of the software, depends on usage even after accounting for code complexity measures, which are known to have an impact on the number of software faults [38, 39, 40, 41, 42], and (2) Deriving a measure for the perceived quality of a software that is independent of its usage.

We used a post-release dataset for an industrial software application in the telecommunication domain that contained information about the number of software crashes and its usage for addressing the research goals, and tested the external validity of our theory, which we developed using the data from the industrial software, using data for a number of NPM packages using the npm-miner dataset [43].

2.3 Data Description

We describe the data used for this study along with the data cleaning and preprocessing steps in this section. For this study, we looked into two different types of software, which are vastly different in nature. The primary focus of the study is on the commercial software developed by Avaya for mobile applications, which are from the telecommunication domain. We chose the Avaya mobile applications because we had access to the actual post-deployment measures for these. We hypothesized that the number of users is the most important measure of usage, but, as we mentioned earlier, obtaining the actual number of users of a software post-deployment is extremely difficult without a dedicated monitoring tool, and such data is often proprietary. Therefore, having access to the actual usage measures gave us a golden opportunity to study the relationship between software usage and software crashes. However, using this data also had some limitations, e.g. we could not choose the variables being measured, the duration of the measurements, or the applications for which the data is being collected. Being a commercial software, the source code is essentially closed-source, which makes it difficult for us to conduct a through investigation with all variables of interest in one model.

For external validation of the theory developed using this data, and examining if the extent of usage has any effect on the number of observed software faults even after taking the various code complexity measures into account, we looked into the NPM packages, which are open-source JavaScript packages used in web-development. We used NPM for our study because (1) it is one of the largest open-source communities, which makes it a good candidate to be investigated, and (2) it collects the number of downloads for the packages, which is a far better measure than other usage measures like the number of stars or forks of GitHub projects.

In this section, we provide some details about the software being studied, discuss the data source, describe the data, and give details about the data preprocessing steps.

2.3.1 Data on Mobile Applications developed by Avaya

2.3.1.1 Software description

One of the software chosen for this study was Avaya Communicator for Android (currently known as Avaya Equinox®). It integrates the mobile devices of the users with their office Avaya Aura®communications environment and delivers mobile voice and video VoIP calling, cellular call integration, rich conferencing, instant messaging, presence, visual voicemail, corporate directory access and enterprise call logs[1].

Another software we studied was the Avaya one-X®Mobile SIP for iOS, which provides mobile communications for the iPhone, iPod touch, and iPad through a wireless-enabled SIP Avaya Aura®environment combining enterprise features with the convenience of a mobile endpoint for users on the go. The Avaya one-X Mobile®SIP for iOS appears as an end point in the Aura®environment[2].

[1]https://support.avaya.com/products/P1574/avaya-equinox-for-android
[2]https://support.avaya.com/products/P0949/avaya-onex-mobile-sip-for-ios

Avaya is developing large, complex, real-time software systems that are embedded and standalone products. Development and testing are spread through 10 to 13 time zones in the North America, USA, Europe and Asia. R&D department employed many virtual collaboration tools such as JIRA, Git, WIKIs and Crucible. Development teams use Scrum-like development methodologies with a typical 4-week sprint. We consider a 15+ year old software component, the so-called Spark engine. As a software platform, Spark provides a consistent set of signaling platform functionalities to a variety of Avaya telephone product applications, including those of third parties. Spark is a client platform that provides signaling manager, session manager, media manager, audio manager, and video manager. The codebase involves more than 200K files and, over all forks, over 4M commits. The Android software chosen for this study is a fork of the Spark codebase. A more in-depth description of the development process is provided in [44].

2.3.1.2 Data Description: Source

The post-deployment data for the mobile applications were obtained from the Google Analytics platform. Google Analytics is a web analytics service offered by Google that tracks and reports website traffic. It is now one of the most widely used web analytics services on the internet. In addition to traditional web applications it also allows tracking of mobile applications. To do that, the producer of a mobile application needs to set up an account and instrument their mobile application to send certain events to Google Analytics. Notably, it works for the mobile applications investigated in this study.

We collected data for a number of mobile applications developed by Avaya from Google Analytics, but some of the datasets turned out to be unusable for this study, for reasons ranging from very low volume of collected data (*e.g.* Avaya Communicator for Android - Experimental Releases) to zero recorded exceptions making an analysis impractical (*e.g.* Avaya One-X®ScsCommander). The following datasets were found usable:

- Avaya Communicator for Android - General Availability and Development versions.
- Avaya one-X®Mobile SIP for iOS - General Availability versions.

The data was collected between December 2013 and May 2016, although the exact time varies across the applications. Although we are primarily focused on the General Availability (GA) versions, since only these versions are available for end-users, we also decided to look into the development version for Avaya Communicator for Android, since we have detailed data available for these versions and we wanted to see if it shows a different characteristics from the GA versions.

Table 2-1. Measures available in the Original Data

Application Release Version	No. of exceptions†
Operating System version in the user's device	Date of record entry
No. of fatal exceptions†	No. of new visits†
No. of visits†	Time on site†
Details on user's device: brand, category (mobile/ tablet) and model	No. of new users†
No. of total users†	Sessions per user†

The original data obtained from Google Analytics had measures for the variables listed in Table 2-1, aggregated at a per-day granularity, meaning that each entry in the original data table contained the measures for the numerical variables (marked with a † symbol in the table) for each unique combination of date, application release version, operating system version, mobile device brand, category, and model. We had the same set of variable for all the applications listed above. As we mentioned earlier, we had no role in selecting which variables to measure, and we received the data "as-is".

It is important to note that Google Analytics releases only aggregate data even to developers of the application and limits the number of REST API calls, so one can not, for example, retrieve usage data for every calendar second or get exact time of the events. The daily counts split by release version of the application, OS version, and type of device, provided sufficiently fine granularity for our analysis.

2.3.1.3 Data Preprocessing

This section contains the data cleaning, transformation, and variable construction steps undertaken prior to the application of the different modeling methods. The major preprocessing step is aggregating the measures to a per-release granularity. We had two main reasons for aggregating the data:

1. The goals of our study are concerned with identifying the relationship between exceptions and other post-deployment variables for different releases, and defining a quality measure to compare the qualities of different releases. Therefore, having the measures aggregated at per-release granularity is essential.
2. We would have been able to take a time-series based approach and still work out our goals if the releases were cleanly separated in time, i.e. if there were no overlap between releases. Unfortunately, we observed from the data that users continue to use one release long after the subsequent releases are available, and there is no clear

pattern about how long a release is used. Therefore, we had to aggregate the data to a per-release level to be able to achieve the goals of this study.

The preprocessing steps we took are discussed below:

Removal of variables before aggregation: Upon initial investigation into the data, we found that no. of exceptions and no. of fatal exceptions were exactly the same, as recorded by Google Analytics, so we removed the no. of fatal exceptions from the dataset. Only fatal exceptions were recorded for this application, i.e., crashes that require a complete restart of the mobile application and, potentially, may affect the operating system itself. This is not surprising since the bulk of the functionality for the application was written in C++ and called from Android Java applications via Native Interface. We did not consider the variables related to mobile device details and Android operating system versions because the application, as noted above, was primarily written in C++ and the user interface aspects that vary greatest among devices and versions of OS were not likely to have influence. To validate that assumption we investigated and found no correlation of exceptions with either variable.

Aggregating data to per-release granularity: We had some missing values in the data, however, most of the missing data was about the mobile devices and since we didn't use them in our analysis, we got rid of that problem by simply dropping the variables. Since our aim is to model the quality of the different releases, we aggregated the data to a per-release granularity, from the the original data that was recorded in per-day granularity. The raw data contained 177 different GA releases and 25 development releases for the Avaya communicator for Android and 11 GA releases for the Avaya mobile SIP for iOS. We dropped 4 GA releases for the Avaya communicator for Android from further consideration because a significant portion of observations were missing. The result of aggregation, however, was two new variables: start date (first day for which we have a record for that release) of a release, and end date (last date for which we have a record for that release) of a release, which in turn helped create another variable: duration of a release. We did not to keep the end date in the final table, since duration and start date can be used to compute the end date.

Verifying the correctness of Release date: 6

The original data involves only the usage aspects and the version information of the software. The project under consideration was relatively new and it was one of the early attempts for the team to deploy mobile software on Android and iOS. As such, not everything was well documented and also was rapidly evolving over time and no record of the exact release dates for most of the releases was available. We did manage to get release dates for some of releases from Google Play Store/ Apple App Store, but not all the release dates were available. For the releases with dates available on Google Play Store/ Apple App Store, the official release dates from Avaya records, and the start dates obtained from the data were either very close or exactly the same, so we do not have a reason to doubt the dates obtained from the data.

Removal of variables post aggregation: The numerical variables were aggregated to give a sum for each variable. Upon further inspection, we found the number of users, new users, visits, and new visits to be highly correlated. In the second iteration, we removed the variable "sessions per user", because aggregating it directly is meaningless, and we were not sure how it was originally calculated by Google Analytics (was it a mean or a median? were new users or total users counted?). We also removed the "total users" and "total visits", because while summing up the new users/visits for each day gives an accurate measurement of the total number of new users/visits for a release, it is not guaranteed that summing up total users/visits does the same due to possible double counting the number of users/visits.

Final list of variables: Keeping the goal of our study in mind, the variables we have after the initial cleaning steps give us necessary information for a model of post-release defects and software usage. In our list of variables, we have the total number of exceptions *i.e.* post-release defects. As for measures related to software usage, we have the total number of new users; the "Time.On.Site" variable, normalized by the number of users of a release, provides a measure for the temporal intensity of usage per user; and the number of visits per user is a measure for the frequency of usage. We also have two variables related to each individual release: the start date *i.e.* the release date gives a measure for the calendar time of each release, and is useful in gaining insight about if the number of post-release defects and software usage vary with time, and the duration of a release, which could have an effect on the number of exceptions and the number of new users, since these variables were not normalized with duration. Since we only have a limited amount of data, we restricted ourselves to use only these six variables. Our final aggregated data table had the measures listed in Table 2-2, with the corresponding variable names we used in the model enclosed in brackets.

To reiterate what we mentioned earlier, we had no control over which variables to measure during data collection, however, while the set of variables we obtained are not

Table 2-2. Measures in the Aggregated Data Table

Release variable - Start Date for the release (Release.Date)	*Release variable* - Effective Duration of the release (Release.Duration)
Post-Release defects - Total No. of exceptions (Exceptions)	*Usage variable* - Average time on site per user (Usage.Intensity)
Usage variable -Total number of new users (New.Users)	*Usage variable* - No. of visits per user (Usage.Frequency)

exhaustive, we believe the three usage related variables: number of new users, usage intensity, and usage frequency adequately capture and report how much usage a software is getting.

Log-transformation of variables: The release date was converted from the Date format to numeric format, which resulted in the values for the release date variable being represented by the difference in days from Unix time (counted from 1970-01-01). We found that all of the variables under consideration had a long-tailed distribution, so we took logarithm of them. The distribution of the variables of GA releases of Avaya communicator for Android is shown in Figure 2-1. The distribution of the variables of other applications is available in our GitHub repository: https://github.com/tapjdey/release_qual_model.

2.3.2 The NPM Packages

As mentioned before, we looked at 520 NPM packages for examining the interrelationship between the code complexity measures, the extent of usage, and the number of issues. The code complexity measures for these packages were obtained from the npm-miner dataset [43], which contained information on 2000 packages (one particular release for almost all packages). However, we decided to only look at the package releases which were released more than a month before the data was collected, because we used the number of downloads over a month as our measure of usage (since daily or even weekly download numbers tend to be quite noisy), and we ended up with 520 releases of 520 different packages (one particular release per package).

However, for answering our fourth research question about exploring how our quality measure varies with time for the different NPM packages, and how it compares to the number of issues, the direct measure of observed software faults, we decided to broaden our scope to look at all NPM packages with more than 10,000 downloads per month (according to [45], automated downloads are expected to be around 50 per day, or 1500 per month,

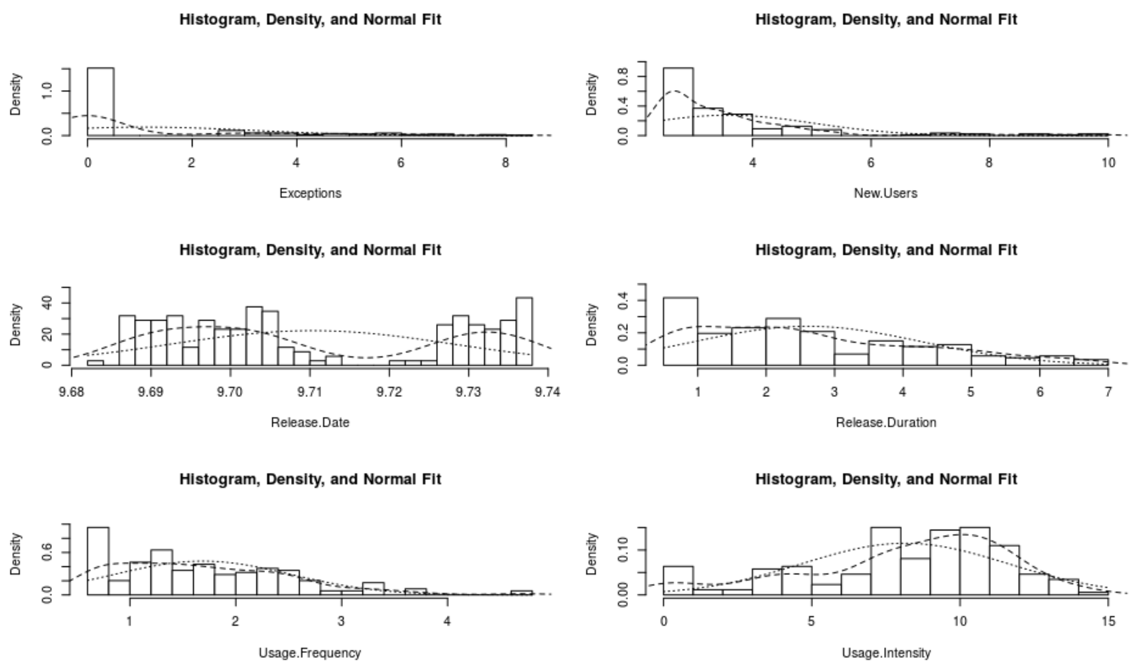


Figure 2-1. Distribution of the variables after transformation:
GA releases of Avaya communicator for Android

and packages with over 10K downloads should, therefore, not be noticeably impacted by downloads by automated sources.) and a GitHub page with issues. With this criteria we ended up with 4430 packages, which contained the 520 packages we used for analysis earlier.

2.3.2.1 The NPM Ecosystem

Node Package Manager or NPM is one of the most active and dynamic software ecosystems at present. It hosted more than 800,000 packages at the time of data collection, and have more than doubled in size in past couple of years (in January 2017, NPM reportedly hosted around 350,000 packages [46]). The popularity of NPM packages have, accordingly, skyrocketed as well. According to [47], “JavaScript is getting more popular all the time, and NPM is being adopted by an ever greater percentage of the JavaScript community." About 75% of all JavaScript developers used NPM, with about 10 million users, in January 2018, according to [47]. Therefore, NPM is an excellent candidate for this study. Moreover, since they track the number of downloads of all packages in the ecosystem, which, in spite of essentially being a mix of downloads by users, bots, and mirror servers, as explained in [45], is the closest measure of usage we could find for open-source projects, and is a far better measure than, *e.g.* number of stars of a GitHub repository which was used in studies like [48] as a measure of popularity, which is little different from usage as we measure it.

2.3.2.2 Measures collected from npm-miner dataset

The npm-miner dataset [43] contained information on 2000 NPM packages with data from the following tools and APIs: *(1) eslint, (2) escomplex, (3) nsp, (4) eslint-security-plugin, (5) jsinspect, (6) sonarjs, (7) npms.io,* and *(8) GitHub*. As mentioned before, we used 520 packages for our analysis in this study. We used the monthly download numbers, collected from the analysis result of *npms.io* and the code complexity measures, collected from the analysis result of *escomplex*[3]. In particular, we looked at the following code complexity measures for each NPM package, since they represent the average per function complexity measures for the packages:

- *loc*: The average per-function count of logical lines of code.
- *cyclomatic*: The average per-function cyclomatic complexity.
- *effort*: The average per-function Halstead effort.
- *params*: The average per-function parameter count.
- *maintainability*: The average per-module maintainability index.

2.3.2.3 NPM data: Defining collection parameters

We found 4430 projects which had more than 10,000 monthly downloads since January 2018 and also had public GitHub repositories with nonzero number of issues. We collected the number of downloads and the total number of issues for all these packages from 2015-03-01 to 2018-08-31. However, we did not conduct a release by release comparison for these packages, because the release durations vary by a lot for most packages. Since the recorded number of downloads is a mix of downloads by human and non-human users, a release by release comparison would not give a reliable picture of the effect of actual usage by human users on the number of issues. However, the number of downloads by bots are relatively stable and vary only with time [45], so controlling for the date (time variable) would eliminate the spurious effects of downloads by bots. So, we decided to

[3]https://www.npmjs.com/package/escomplex#result-format

focus on the entire packages instead of releases of the packages, and measured the effect daily downloads have on number of issues of that package on that day after controlling for the calendar date.

2.3.2.4 NPM data: Data collection for the 4430 packages

We used the API provided by NPM for collecting daily downloads of the 4430 NPM packages. (The API documentation is available in:
https://github.com/npm/registry/blob/master/docs/download-counts.md).

To obtain the metadata information for every package in NPM, we wrote a “follower" script, as described in
https://github.com/npm/registry/blob/master/docs/follower.md. The output contained the metadata information for all releases of all packages in NPM. From this we extracted the URL of GitHub repositories of the packages. Some NPM packages do not have a valid GitHub URL, so those were dropped from subsequent analysis, as per the criteria we defined. Using the Rest API provided by GitHub we collected information on the issues for all these NPM packages. We collected the total number of issues for all these packages from 2015-03-01 to 2018-08-31.

Finally, we used the issue creation dates to construct a dataset of the total number of issues per day. We used the total number of issues instead of the number of open issues because we are interested in the number of issues encountered by the users of the packages. Whether an issue is resolved or not depends on a number of factors, *e.g.* the number of developers, the responsiveness of the developers, the number of packages managed by each developer, the complexity of the problem; most of which are unrelated to usage, so we decided using the total number of issues is a much more reasonable option. This same issue data was used by our analysis of the 520 NPM packages, where we used the number of issues created for those packages during one month prior to the date the npm-miner data was collected.

2.3.2.5 NPM data preprocessing

For the analysis of the 520 NPM packages, we constructed a dataset containing the 5 quality measures of the packages (variable names: *loc, cyclomatic, effort, params,* and *maintainability*), the number of downloads (variable name: *downloads1M*) during the month before the data collection date for npm-miner dataset (2018-01-22), and the number of issues created for those packages during the same time. The variables were log

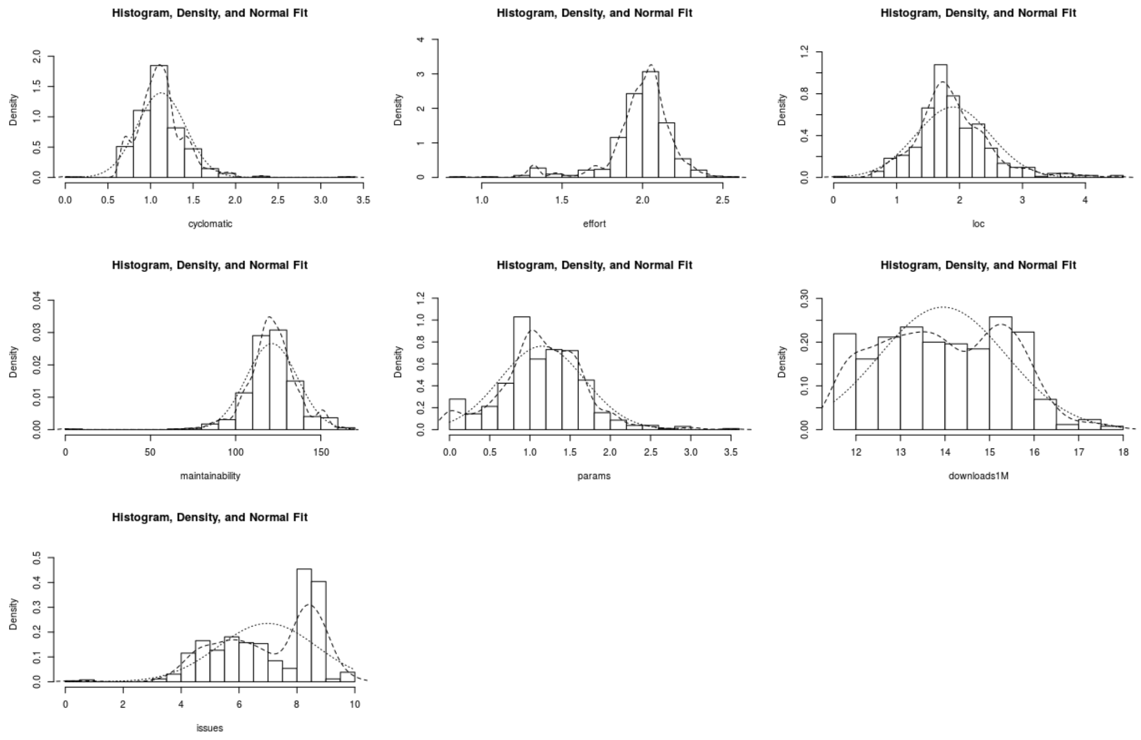


Figure 2-2. Distribution of the variables after transformation: the 520 NPM packages

transformed to correct the skewness of the data, except *maintainability* and *effort*, since these variables were not skewed. The distribution for the transformed data is shown in Figure 2-2. For the purpose of applying BN models, the data was scaled as well.

The data for the 4430 NPM packages which was used to compare the trends had the calendar date, the cumulative number of issues for the packages until that date, and the number of downloads on that date. Since this data was used for demonstrating the trends, we did not transform this dataset.

2.4 Methodological Overview

In this section we describe the methodology we followed in this paper. We employed two different modeling techniques for finding out the relationship among the post-release variables: (1) Bayesian structure search method, and (2) Random Forest Regression method. Since we primarily focus on finding which variables have the most impact on the number of exceptions (or issues for the NPM packages), we used it as the response variable for the Random Forest regression model.

We considered using the OLS estimator since it is one of the simplest modeling methods that gives good models in a lot of situations and the result is easy to interpret. However, we were unsure about the accuracy of the result due to the presence to moderate to high correlation between some of the predictor variables (e.g. the Release Date and Release Duration variables had a correlation of -0.88 for the iOS application data). Moreover, we found that our variables do not satisfy all the criteria (laid out by the Gauss–Markov theorem) for creating the best linear unbiased estimator (BLUE), so we ended up not using it in our study. Instead, we decided to use Bayesian Network (BN) for modeling the interrelationship among these variables, since the accuracy of this model is unaffected by the presence of high correlation among the predictors. Variables with high correlation simply appear as connected nodes in the final model. This eliminates the need of dropping some of the correlated variables from the model, which introduces subjectivity during the modeling process. Since the use of Bayesian Network models is not very common in this context, we discuss BN models in greater detail later in this section. The other modeling approach we used is Random Forest regression method. Random Forest is one of the best off-the-shelf models that works well with almost all types of data and generally does not overfit, and it is easy to get the relative importance of the predictor variables from a fitted model. These two factors led us to use Random Forest regression as the other modeling technique to identify the most impactful predictors explaining the number of exceptions. To find the best fitting Random Forest model, we performed a grid search using the "tune" function of the "e1071" R package to find the best model parameters "ntree": the number of tress to grow, and "mtry": the number of variables randomly sampled as candidates at each split. Since the sample size of datasets are limited, we used 10 times 2 fold cross-validation as the tuning method.

2.4.1 Bayesian Network Models

Bayesian Network [49, 50] is a type of Probabilistic Graphical Model (PGM), which explicitly represents the conditional dependency/independence as a directed acyclic graph where variables represent nodes and dependencies represent links, and thus this representation can be used as a generative model[4]. Bayesian Networks models can

[4]A generative model specifies a joint probability distribution over all observed variables, whereas a discriminative model (like the ones obtained from regression or decision trees) provides a model only for the target variable(s) conditional on the predictor variables. Thus, while a discriminative model allows only sampling of the target variables conditional on the predictors, a generative model can be used, for example, to simulate (i.e. generate) values of any variable in the model, and consequently, to gain an understanding of the underlying mechanics of a system, generative models are essential.

be useful in the context of Software Engineering research [36] due to having several advantages over regression models. To be precise, regression analysis is a very simple BN where there is one directed link from each independent variable to dependent variable. BNs, therefore, can help with multicollinearity, a common problem with software engineering data [51, 52, 53, 54], that is present in our data as well, by linking independent variables.

Another variety of PGM that we did not use in this paper (details in Section 2.8) is the Markov random fields that represent the interrelationships between variables as undirected graphs. They differ in the set of independencies they can encode and the factorization of the distribution that they induce [49].

Bayesian Network Model construction: Despite the promises of BNs, they tend to be quite sensitive to data, and operational data, is often problematic [55, 56]. Careful preprocessing, therefore, is needed to ensure a reliable and reproducible result. Two primary ways to use BNs exist. With the first approach the graph represents dependencies obtained from domain experts. The graph may include prior distributions about the parameters of the overall model. The data is then used to calculate the posterior distribution and to make inference. The second approach puts minimal a-priori assumptions about the model and focuses on the search for the best graphical representation for a given dataset (structure learning). This is an NP-hard problem [57], but a number of different heuristic structure learning algorithms are available. Due to the lack of any strong theory connecting the variables we are considering, we decided to use the structure search method for BN model construction. Since our goal is to find a Bayesian network model for the data, we didn't examine the methods that do not result in a Directed Acyclic Graph (DAG). We found that the *bnlearn* package in R implements a wide range of BN searching methods for continuous, discrete, or a mixed set of variables and the corresponding families of scoring functions and also has a good number of examples. These methods were also shown to be able to recover the underlying network for a protein-signaling-chain (in Biology) in [58]. We, therefore, use this package for our analysis. In addition to the methods implemented in *bnlearn* package, we investigated some methods from a few other packages which can be interfaced with the *bnlearn* package.

Due to the potential inconsistencies of the BN models, we performed our modeling in two stages. First, we considered all available BN structure methods in the *bnlearn* package and ran a simulation based study to find the methods that are most accurate and then we used those methods on our data to create the final model.

Methods considered:

The different BN structure search methods we considered are listed below:

- *Greedy Hill-Climbing search algorithms*(HC) [59, 58]
- *Hybrid algorithms*(Hybrid) [59, 58]
- *Posterior maximization* using *deal* package in R [58, 60] .
- *Simulated Annealing* using *catnet* package in R [61, 58].
- *PC Algorithm* using *pcalg* package in R [62, 63, 58].
- *MAP (maximum a-posteriori estimation) Bayesian Model Averaging* (MAP) [59, 58]

This is not an exhaustive list of all possible BN structure search methods, in fact, it is impossible to make an exhaustive list for a heuristic search method like this, however, they represent a class of popular heuristic structure search methods that are part of the “bnlearn” package, which is a popular R package that is in continuous development since 2007.

All structure search algorithms try to maximize some form of a network score. Among the various scores available, BIC score is the suitable one when the goal is to create an explanatory model from non-informative prior models [64, 65]. BIC score is used for discrete data while the Gaussian equivalent of BIC (bic-g) score is used for continuous data.

The results, *i.e.* the structure and the parameters resulting from a structure search algorithm, are often noisy, meaning that different settings induce slightly different networks. To mitigate this effect we use non-parametric bootstrap model averaging method described in [66], which provides confidence level for both the existence of edge and its direction. This enables us to select a model based a confidence threshold. Authors of [66] argue that threshold is domain specific and needs to be determined for each domain. For instance, a threshold of 0.95 indicates that only the edges that appeared in more than 95% of the bootstrap optimized models were selected.

Many applications of BNs discretize the data prior to applying the structure learning methods, and in some cases where the data distribution is too skewed to fit the normality assumption, discretizing the data produces better models than using continuous data, so we considered it as a possibility as well.

Using continuous data works best when the random variables (possibly after a transformation) have Gaussian distribution. While using discrete data does not require

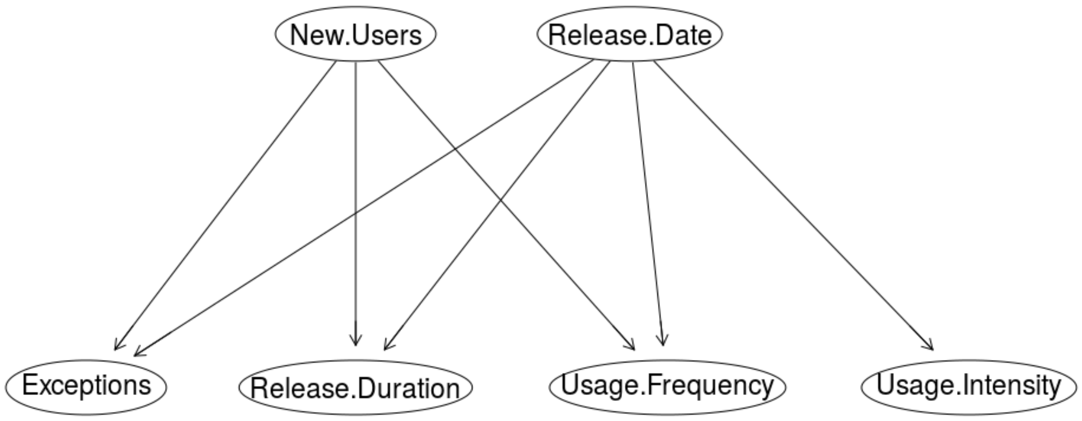


Figure 2-3. Custom model used for Simulation Study

such assumptions, obtaining the optimal discretization for a dataset is in itself an NP-hard problem [67]. Choosing a sub-optimal discretization technique may result in spurious or missed relationships, which can in turn result in incorrect dependencies being reported in the resulting model. Given the pros and cons of both types of methods, we use methods of both types for our simulation study. As we are interested in creating a generative model, we had to use a discretization method that is unsupervised. The basic problem with commonly used supervised methods (*e.g.* Chi-square, or MDLP discretization algorithms) is that they optimize discretization to improve explanatory power for a single response variable. This is not suitable for a BN structure search, because we do not know which variables will be responses (have arrows pointing to them) and which will be independent (have no incoming arrows) *a-priori*. While some research on multidimensional discretization methods exists [68], we are not aware of any that have a robust implementation.

Simulation Study:

We performed the simulation study by first creating a random BN (see Figure 2-3) with six nodes, since we also have six variables in our final list (Table 2-2). For demonstration purposes we use the same variable names. We fitted this graph with our data on GA releases of Avaya communicator for Android (log-transformed and scaled) to generate values for the coefficients for each edge. This model was used in our simulation study going forward. We created 1000 different simulated datasets from the BN structure in Figure 2-3, and applied the different structure search algorithms (both continuous and discrete versions, where available) listed above. Our performance metric is finding how many times the different algorithms can recover the underlying structure from the simulated data.

Other than testing the methods themselves, we also tested whether or not we should discretize the data. We tried different discretization methods, *viz.* equal interval, equal frequency, and k-means clustering based discretization methods from the *arules* package [69], and the Hartemink[5] discretization methods in the *bnlearn* package.

Except for the *Posterior maximization* using *deal* package, which can't be bootstrapped, all other results were bootstrapped, so we tested different thresholds in our simulation study as well. Finally, for the the Hybrid search algorithm, in which conditional independence tests are performed to restrict the search space for a subsequent greedy search, there are many restrict methods available, *viz.* gs" (Grow-Shrink), "iamb" (IAMB), "fast.iamb" (Fast-IAMB), "inter.iamb" (Inter-IAMB), "mmpc" (Max-Min Parent Children), "si.hiton.pc" (Semi- Interleaved HITON-PC), "chow.liu" (Chow-Liu), "aracne" (ARACNE) [71], and we tested all of these restrict options in our simulation study.

The result of the simulation study is shown in Table 2-3, which shows the fraction of times exact structures and off-by-one structures[6] were generated by each method in the simulation. The result varies with the chosen threshold, so in Table 2-3, we show the overall performance of the different methods which generated an exact or off-by-one structure at least once in the simulation. For the hybrid search methods, we list mention the restrict option that was used, and the '-D' suffix indicates a discretization method was used to discretize the data prior to applying a structure search method. '-D-H' indicates Hartemink discretization method and '-D-F' indicates Equal-Frequency discretization method. It is clear from the table that only HC and MAP methods can effectively reproduce the correct underlying structure around half of the times and they create more off-by-one structures than others, indicating the error rate is the lowest for these methods.

In Table 2-4, we show the fraction of times exact and off-by-one models were generated by HC and MAP methods, which performed the best among the methods considered, for different thresholds. It can be seen that using a moderately high threshold between 0.75 and 0.9 gives good results for both HC and MAP, while higher thresholds for HC and lower

[5]Hartemink's pairwise mutual information method[70].

[6]one extra / missing / reversed edge

Table 2-3. Result of Simulation Study

Method	Exact	Off-by-one
HC	0.574	0.264
MAP	0.596	0.214
Hybrid- si.hiton.pc	0.000	0.019
Hybrid- mmpc	0.000	0.016
Hybrid- gs	0.000	0.011
HC-D-F	0.000	0.010
Hybrid- iamb	0.000	0.010
Hybrid- mmpc -D-H	0.000	0.008
Hybrid- si.hiton.pc -D-H	0.000	0.008
HC-D-H	0.000	0.007
Hybrid- mmpc -D-F	0.000	0.007
Hybrid- si.hiton.pc -D-F	0.000	0.006
Hybrid- iamb -D-F	0.000	0.005
Hybrid- gs -D-F	0.000	0.004
Hybrid- gs -D-H	0.000	0.004
Hybrid- iamb -D-H	0.000	0.002

Table 2-4. Result of Simulation Study: Different Thresholds

Method	Threshold	Exact	Off-by-one
MAP	0.85	0.68	0.25
MAP	0.80	0.67	0.25
MAP	0.90	0.67	0.26
MAP	0.95	0.66	0.27
MAP	1.00	0.66	0.27
MAP	0.75	0.66	0.21
HC	0.65	0.63	0.23
HC	0.70	0.63	0.23
HC	0.75	0.63	0.23
HC	0.80	0.63	0.23
HC	0.85	0.62	0.24
HC	0.55	0.62	0.23
HC	0.60	0.62	0.23
MAP	0.70	0.62	0.21
HC	0.90	0.60	0.26
MAP	0.65	0.58	0.17
HC	0.95	0.57	0.29
MAP	0.60	0.43	0.14
MAP	0.55	0.33	0.11
HC	1.00	0.19	0.47

thresholds for MAP give worse results. Using the optimal threshold creates models that have more than one wrong and/or missing edge only 7-14% of the times.

The result of the simulation study had the following findings:

- Using structure search algorithms on the continuous data resulted in much more frequent recovery of the original BN structure compared to discretized data.
- Bootstrapping improves the stability of the results considerably.
- The bootstrapped Hill-Climbing search and MAP Bayesian Model Averaging algorithms outperformed all others both in terms of accuracy and runtime, being able to recover the underlying structure more than 63% of the times and making no more than one error 86% of times with optimal thresholds.

2.5 Results and Analysis

2.5.1 Modeling the relationship between Exceptions and other post-release variables

As mentioned earlier, we conducted our analysis in two stages: first, we used Bayesian Network (BN) modeling approach to identify the interrelationship between the variables and then, we used a random forest (RF) model to verify the results.

2.5.1.1 Bayesian Network Model

One key assumption for applying the continuous BN structure search algorithms is that the variables have a distribution close to a Gaussian distribution. To satisfy this modeling assumption, we scaled all the variables to unit scale. The variable “Exceptions" still had a long tailed distribution, but the distributions of the other variables were much closer to normal distribution.

According to the result of the simulation study, we decided to use bootstrapped hill-climbing search and MAP Bayesian model averaging methods for constructing the Final BN models for our datasets and considered the model that resulted from both the methods. The resultant BN model for the GA releases of Avaya Communicator for Android is shown in Figure 2-4, which shows “New.Users” and “Release.Date” are parent nodes of “Exceptions”. Figure 2-5 shows the final BN Model for Development releases of Avaya Communicator for Android, in which only “New.Users” is the parent of “Exceptions”, and Figure 2-6 shows the final BN Model for GA releases of Avaya mobile SIP for iOS, where once again “New.Users” and “Release.Date” are parent nodes of “Exceptions”. In these figures p-values $< 2e - 16$ are denoted as 0.

Every bootstrap run was performed over 500 bootstrap samples, and a hill-climbing search with 100 random restarts was applied on each sample to find the best fitting network, so in essence, each resultant network was obtained by averaging 50,000 candidate networks. We used a Threshold of 0.85, as it seemed optimal from our simulation study.

The result form a bootstrap run shows the relative strength of the link and the relative confidence for the direction of the link. In Table 2-5 we have shown the result from one bootstrap run of the HC method for all possible edges for the GA release data of Avaya Communicator for Android. If an edge has $< 50\%$ confidence in its direction, then the edge appears in the opposite direction in our model. Although Bayesian Networks are sometimes interpreted as causal relationships [72], there are disagreements on how that should be done. We, therefore, are not interpreting these relationships as causal here. All

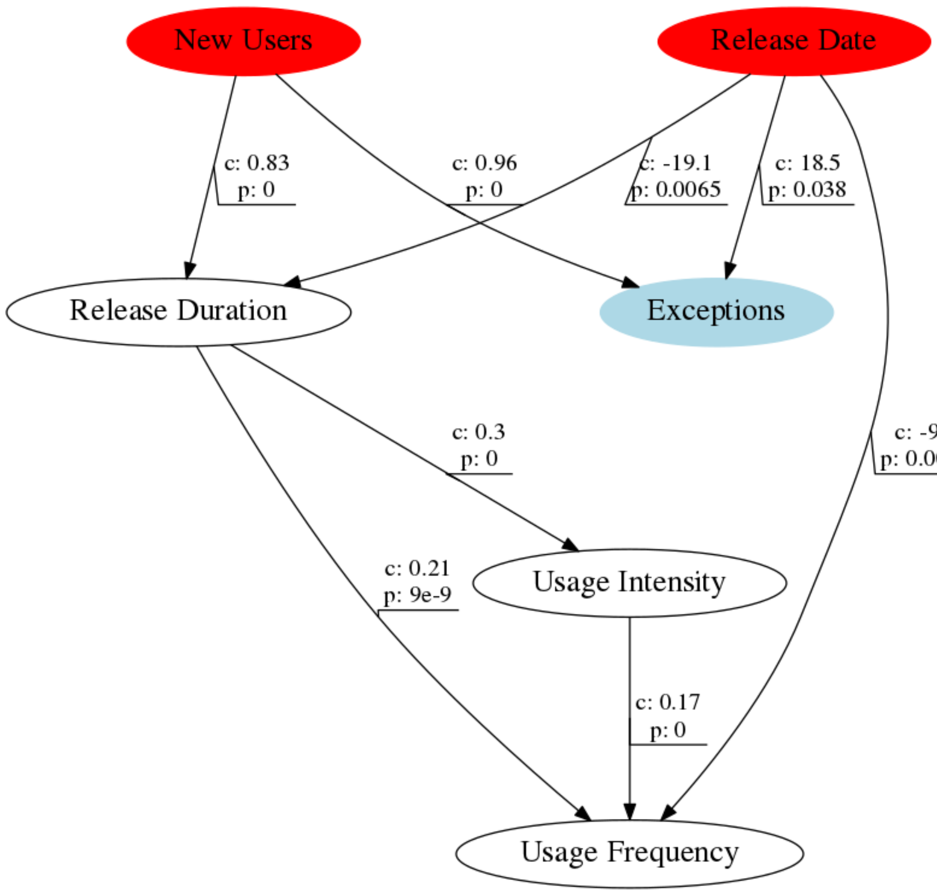


Figure 2-4. Final BN Model for GA releases of Avaya Communicator for Android (with c: coefficients after fitting the transformed, but unscaled data, p: p-value for the link)

observed links, therefore, indicate the presence of observed correlation (and are empirical in nature) and the direction is a property of the topological ordering of nodes in a DAG, and affects the total probability distribution of the variables.

The BN models were fitted to the unscaled data, and the resulting coefficient of each link is also shown in the figures. The p-value for each link was calculated from a linear model with the source nodes as predictors and the destination node as the response variable, e.g. the p-value for the link from “New.Users" to “Exceptions" was calculated by looking at the result of: `lm(Exceptions ~ New.Users + Release.Date)`. We fitted the model to the transformed, but unscaled data (for easier interpretation of results).

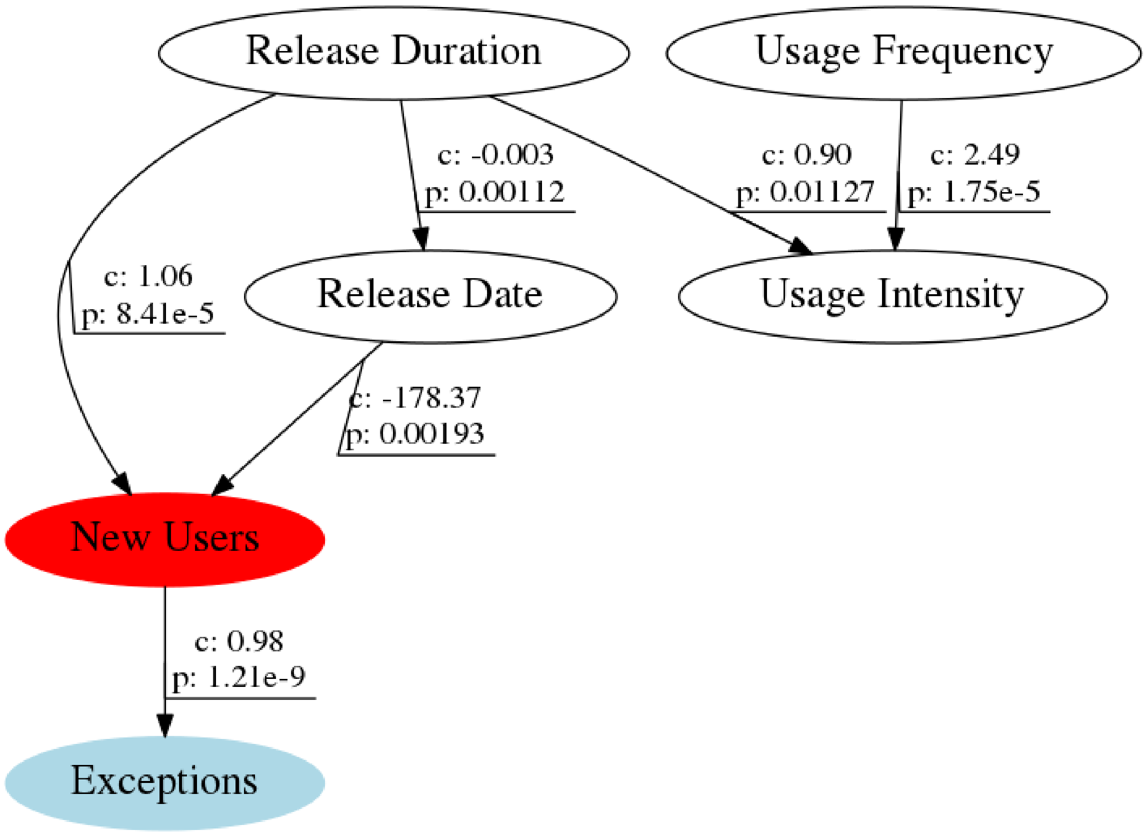


Figure 2-5. Final BN Model for Development releases of Avaya Communicator for Android (with c: coefficients after fitting the transformed, but unscaled data, p: p-value for the link)

By looking at the p-values for the links, we can say that all the links in the BN models are statistically significant. Links having a negative coefficient indicate an inverse relationship between the parent and the child node. The performance of explanatory models is evaluated by the fraction of deviance explained by the model. Our model explains 80.3% and 45.9% of the variation in “Exceptions" (adjusted R^2 value of the model) for development and GA releases for Avaya Communicator for Android respectively and 42.0% for GA releases of Avaya mobile SIP for iOS. This indicates our BN model is statistically significant, but the predictors we used could only explain around half of the variance in Exceptions.

2.5.1.2 Random Forest Model

As a verification step to identify the important variables affecting the number of exceptions, we used a Random Forest model to fit the data, with “Exceptions” as the

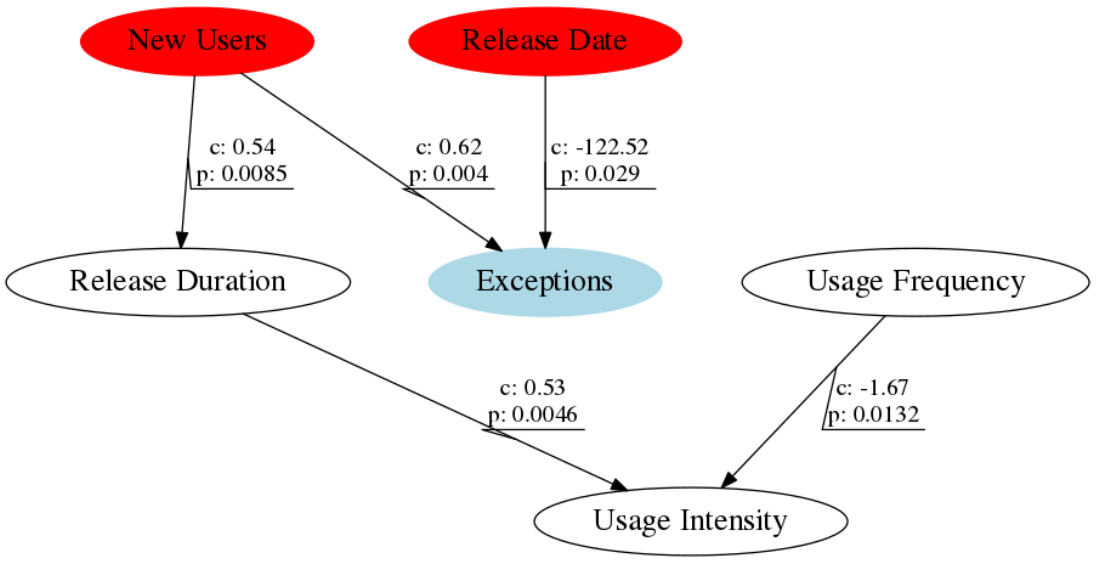


Figure 2-6. Final BN Model for GA releases of Avaya mobile SIP for iOS (with c: coefficients after fitting the transformed, but unscaled data, p: p-value for the link)

response variable. The variable importance plot for the GA release data of Avaya Communicator for Android, as shown in Figure 2-7, indicates that "Release.Date" and "New.Users" are the two most important variables. For the development releases of Avaya Communicator for Android, the variable importance plot is shown in Figure 2-8. "New.Users" is again the most important variable, followed by "Release.Duration". For the GA releases of Avaya mobile SIP for iOS, the variable importance plot again shows the number of new users is the most important variable, as can be seen from Figure 2-9.

The best selected model parameters derived from tuning show that the optimal models were obtained for "ntree"=600 and "mtry"=3 for all datasets. The R^2 values for these models, again obtained from 10 times 2 fold cross-validation, are 0.48, 0.56, and 0.31 for the GA and development releases of the Android application and the GA releases of the iOS application respectively. The poor performance of the iOS application is likely due to the very small sample size of the dataset. Although the overall performance of the models wasn't very good, since we had a limited number of predictors, and none of the internal factors were part of the model, this result shows that even for the purpose of prediction, the number of new users play an important role.

Table 2-5. Example bootstrap result - GA releases of Avaya Communicator for Android

from	to	strength	direction
Exceptions	New.Users	1.00	0.34
Exceptions	Release.Date	0.86	0.47
Exceptions	Release.Duration	0.46	0.50
Exceptions	Usage.Frequency	0.75	0.78
Exceptions	Usage.Intensity	0.35	0.47
New.Users	Exceptions	1.00	0.66
New.Users	Release.Date	0.20	0.62
New.Users	Release.Duration	1.00	0.71
New.Users	Usage.Frequency	0.71	0.85
New.Users	Usage.Intensity	0.34	0.64
Release.Date	Exceptions	0.86	0.53
Release.Date	New.Users	0.20	0.38
Release.Date	Release.Duration	1.00	0.63
Release.Date	Usage.Frequency	0.97	0.82
Release.Date	Usage.Intensity	0.66	0.77
Release.Duration	Exceptions	0.46	0.50
Release.Duration	New.Users	1.00	0.29
Release.Duration	Release.Date	1.00	0.37
Release.Duration	Usage.Frequency	0.90	0.55
Release.Duration	Usage.Intensity	1.00	0.53
Usage.Frequency	Exceptions	0.75	0.22
Usage.Frequency	New.Users	0.71	0.15
Usage.Frequency	Release.Date	0.97	0.18
Usage.Frequency	Release.Duration	0.90	0.45
Usage.Frequency	Usage.Intensity	1.00	0.22
Usage.Intensity	Exceptions	0.35	0.53
Usage.Intensity	New.Users	0.34	0.36
Usage.Intensity	Release.Date	0.66	0.23
Usage.Intensity	Release.Duration	1.00	0.47
Usage.Intensity	Usage.Frequency	1.00	0.78

2.5.2 Deriving a usage-independent measure of Quality

2.5.2.1 Obtaining the Quality measure

In order to arrive at the usage independent quality measure, we follow the framework of establishing laws governing relationships among measures of software development proposed in [73]. Law is an equivalent of invariance, i.e. a function of measures that is constant under certain conditions. In this case we want it to be constant for releases that have the same quality. First, the law requires a plausible mechanism and second, an empirical validation. Each new user may have a different type of phone, operating system,

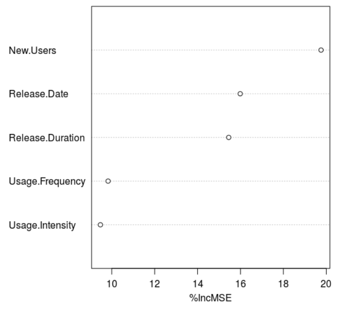


Figure 2-7. Variable Importance Plot of RF model for “Exceptions" for GA release data of Avaya Communicator for Android

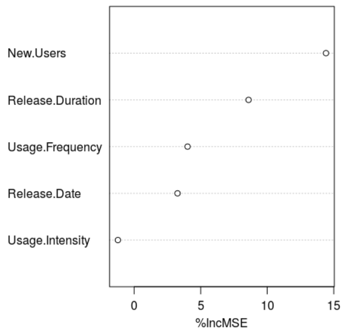


Figure 2-8. Variable Importance Plot of RF model for “Exceptions" for development release data of Avaya Communicator for Android

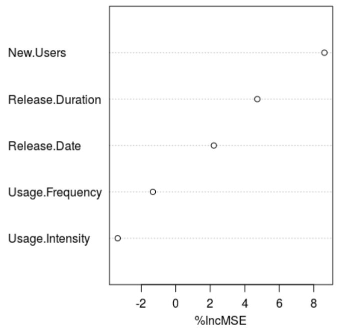


Figure 2-9. Variable Importance Plot of RF model for “Exceptions" for GA releases of Avaya mobile SIP for iOS

service provider, geographic region, and usage pattern. It is reasonable to assume that some of these configurations lead to software malfunction manifested as an exception. This provides us with a plausible mechanism on how precisely more new users of one release might generate more exceptions even if we have two releases of identical quality. We rely on our models (all of which show the number of software exceptions to be dependent on the number of users and on the software release date) to obtain empirical validation of this postulated mechanistic relationship. Therefore, we arrive at the following software law that is applicable for the investigated context: the average number of exceptions experienced by each user should, therefore, be independent of usage and depend only on the qualities of a software release.

In this section we test the above evidence-based hypothesis and provide the result of an analysis with the **number of exceptions per user** as a response variable (“Quality”) representing software quality. *This is actually a measure for faultiness, so a lower value of “Quality" indicates the actual quality of the software perceived by end users is better.*

The value of the “Quality” variable (not log transformed) was seen to be varying between 0 and 10.85 (mean: 0.45, median: 0, standard deviation: 1.48) for the GA release data of Avaya Communicator for Android, between 0 and 22.83 (mean: 1.12, median: 0, standard deviation: 4.55) for development versions of the same, and for the GA releases of

Avaya mobile SIP for iOS it varied between 0 and 0.5 (mean: 0.0488, standard deviation: 0.15) .

2.5.2.2 Establishing the independence of the Quality measure and other usage related variables

Similar to the previous analysis, we applied Bayesian Network search and Random Forest modeling approaches on the dataset containing this quality measure and the remaining variables, all of which were log-transformed.

The result, as expected, shows that the quality of a software, measured by average number of faults experienced by each user, has no dependence on other usage variables. The BN model(Figure 2-10), obtained with a threshold of 0.85 from a bootstrapped Hill-Climbing structure search model, indicates the “Quality” variable depends only on the “Release.Date” variable. Finally, the result of 10 times 2-fold cross-validation with the best RF model (Variable Importance plot in Figure 2-13) with the optimal values for “ntree”(300 in this case)) and “mtry”(1 in this case) indicates that the “Release.Date” variable is much more important compared to others, and the two usage related variables are of much lower importance.

For the development versions of Avaya Communicator for Android, all the predictor variables turned out be insignificant for the BN (Figure 2-11) models. Even the tuned RF model gives a really bad fit in the 10 times 2-fold cross-validation as well, with a R^2 value of -0.42 (the implication of a negative value of R^2 is as explained in [74]), indicating the predictors are very poor. Still, the two usage related variables have the lowest importance in the variable importance plot as seen in Figure 2-14.

Finally, for the GA releases of Avaya mobile SIP for iOS, the BN model (Figure 2-12) shows that release date and release duration have effect on the “Quality” variable, but the two other usage variables have no effect. We did not run RF model on this dataset owing to the very small sample size.

The results from these analyses clearly indicate that the quality measure defined by the number of exceptions per user is independent of software usage, and, therefore, suitable for comparing the quality of software development process among different releases of a software.

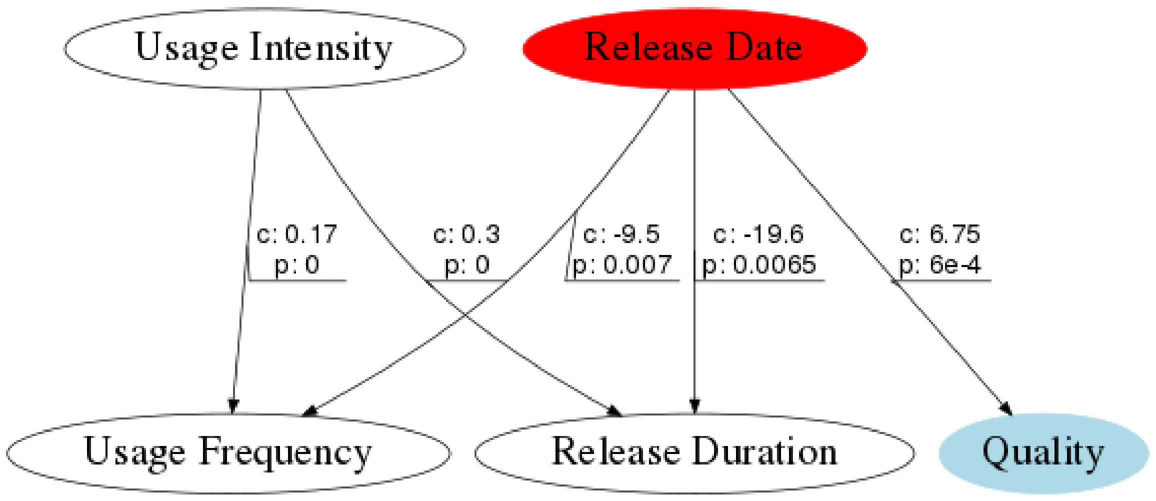


Figure 2-10. Bayesian Network Model for “Quality" - GA releases of Avaya Communicator for Android (with c: coefficients after fitting the transformed, but unscaled data, p: p-value for the link)

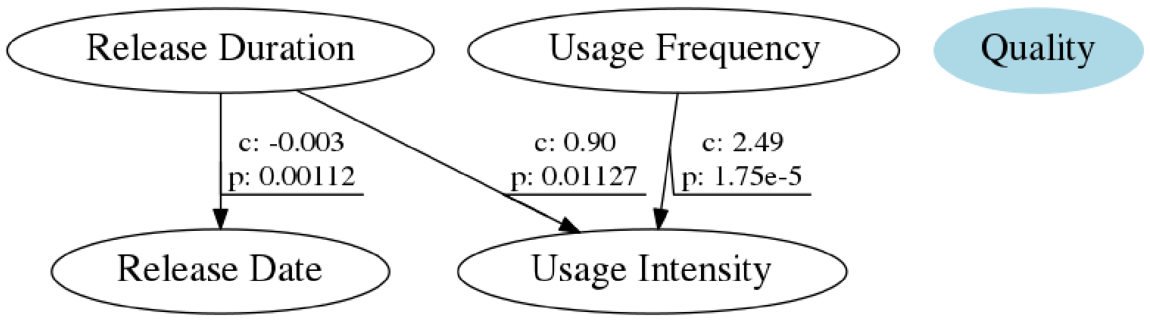


Figure 2-11. Bayesian Network Model for “Quality" - Development releases of Avaya Communicator for Android (with c: coefficients after fitting the transformed, but unscaled data, p: p-value for the link)

2.5.2.3 Timeline of Quality for the mobile Applications

We wanted to see how the perceived quality of the releases of the different mobile applications described above change with time. As a general trend, we observe that most of the exceptions occur right after the release date. then, as the number of users keep increasing with time, the value of the quality variable drop and come to a stable value. In this paper we show only the timeline for GA releases of Avaya mobile SIP for iOS (Figure 2-15), since the other two softwares had a lot of releases, making them difficult to

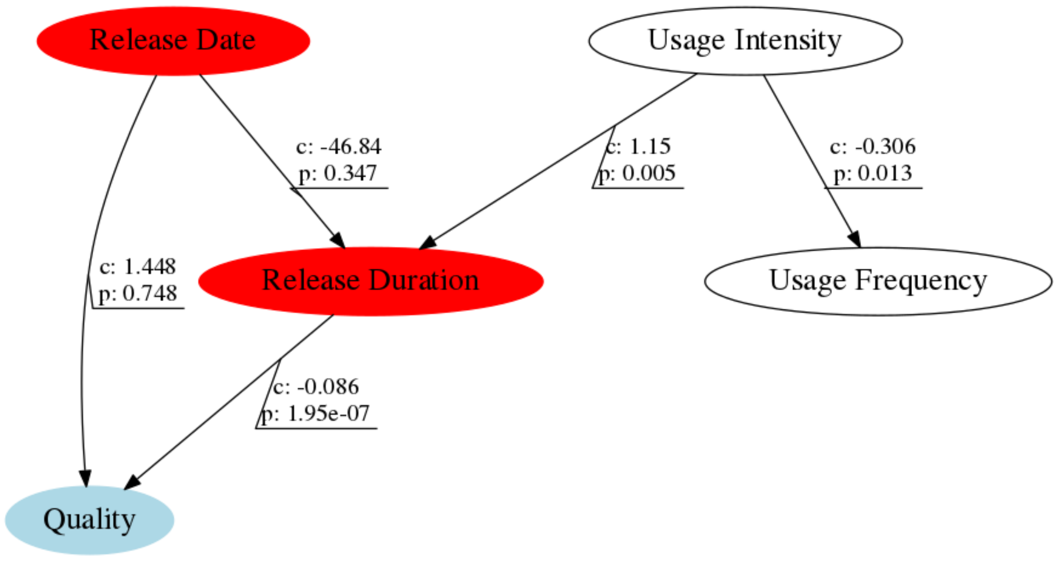


Figure 2-12. Bayesian Network Model for “Quality" - GA releases of Avaya mobile SIP for iOS (with c: coefficients after fitting the transformed, but unscaled data, p: p-value for the link)

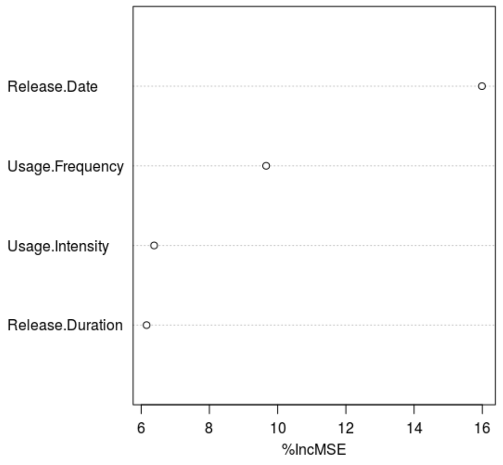


Figure 2-13. Variable Importance plot from the Random Forest Model for Quality Variable - GA releases of Avaya Communicator for Android

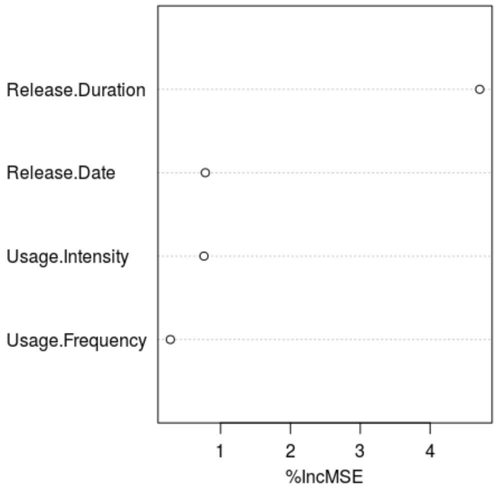


Figure 2-14. Variable Importance plot from the Random Forest Model for Quality Variable - Development releases of Avaya Communicator for Android

identify from the plot. The other two are are available in our GitHub repository: https://github.com/tapjdey/release_qual_model.

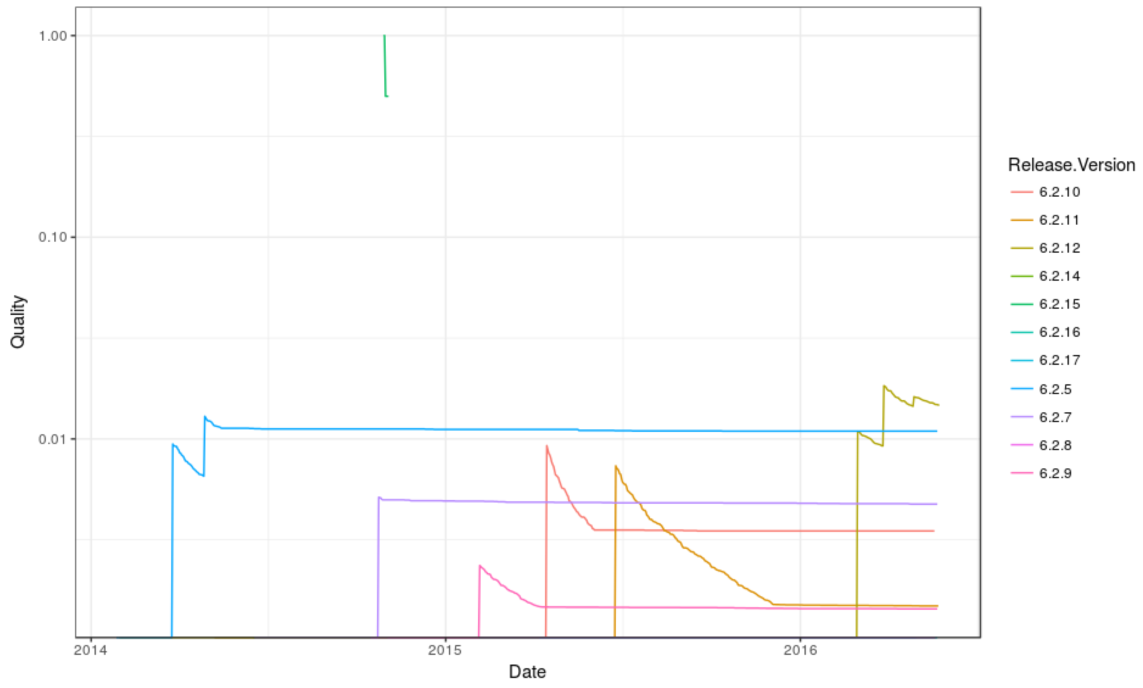


Figure 2-15. Timeline for Quality Variable - GA releases of Avaya mobile SIP for iOS

In Figure 2-16, 2-17, and 2-18, we show the relative trends of Exception and the Quality variable for the GA and development releases of the Android application and the GA releases of the iOS application respectively. We are not interested in the absolute values of the metrics, but the values of the metrics for a release relative to the values for other releases. We only show the releases with non-zero number of exceptions, since if the number of exceptions is zero, the the value of the quality metric is also zero. The blue dotted lines represent the releases dates of different releases, and the black marker and the red cross on the blue line represent the exceptions and the quality variable for that release respectively.

We can see that for a number of releases, Exceptions and Quality follow a similar trend, i.e. if the number of exceptions increase, the value of the Quality variable increases accordingly. However, there are indeed a number of cases where the number of exceptions is relatively small, but the value of Quality variable is larger than that for other releases, e.g. for many of the GA releases for the Android application in 2016, or vice versa, e.g. the release around September 2015 for the GA releases for the Android application. This indicates that if we simply keep on using the number of exceptions as the quality measure, we will misclassify (as being better or worse than other releases) a number of

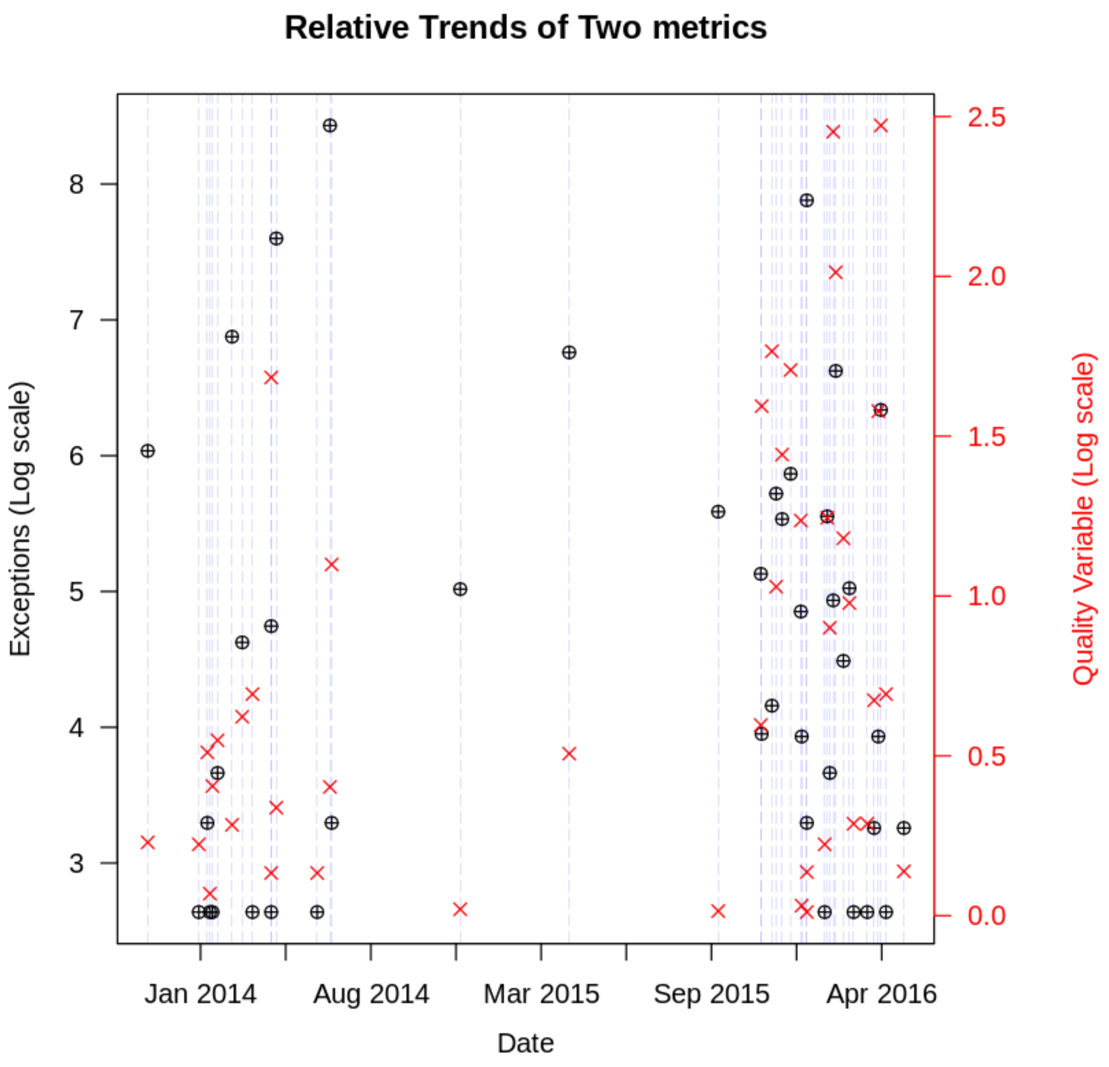


Figure 2-16. Relative trends of Exception (marked with circle) and the Quality variable (marked with cross) - GA releases of Avaya Communicator for Android

releases. This result supports our hypothesis that not accounting for the usage parameter will not systematically misclassify all releases, since the internal factors affecting the release are independent of the external factors, but would randomly misclassify some of them depending on how much usage a release is getting.

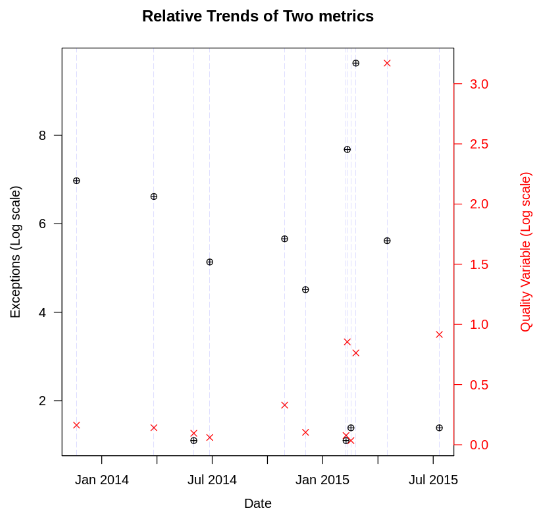


Figure 2-17. Relative trends of Exception (marked with circle) and the Quality variable (marked with cross) - Development releases of Avaya Communicator for Android

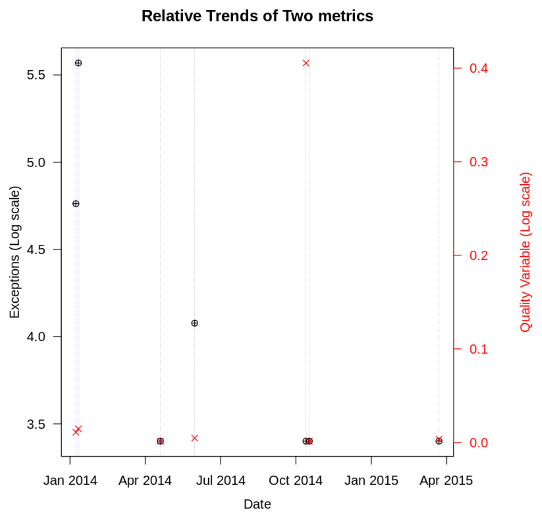


Figure 2-18. Relative trends of Exception (marked with circle) and the Quality variable (marked with cross) - GA releases of Avaya mobile SIP for iOS

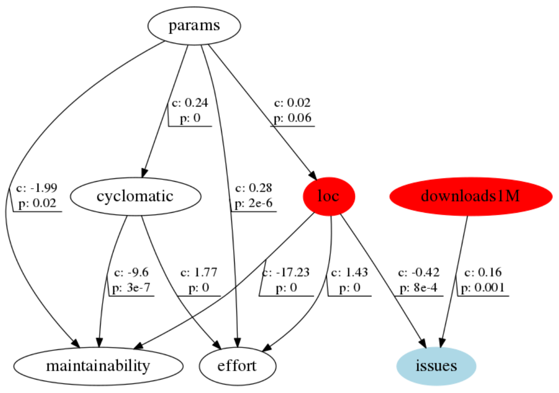


Figure 2-19. Bayesian Network Model for Issues for the 520 NPM packages (with c: coefficients after fitting the transformed, but unscaled data, p: p-value for the link)

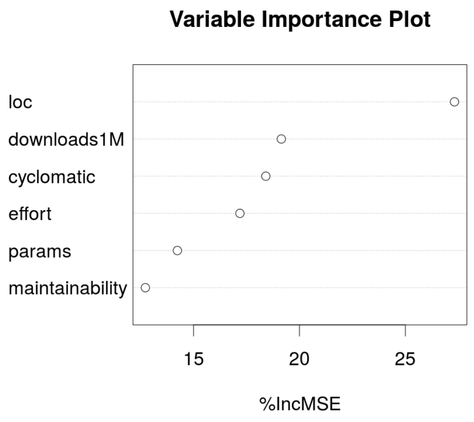


Figure 2-20. Variable Importance plot for the Random Forest Model for Issues for the 520 NPM Packages

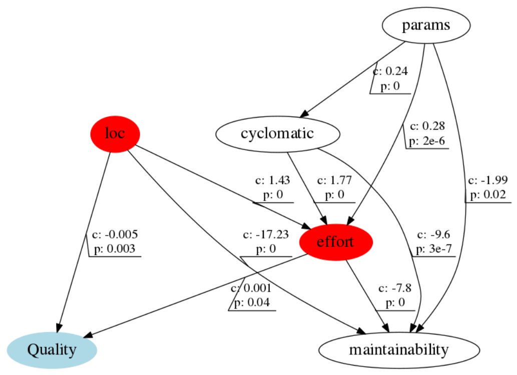


Figure 2-21. Bayesian Network Model for Quality for the 520 NPM packages (with c: coefficients after fitting the transformed, but unscaled data, p: p-value for the link)

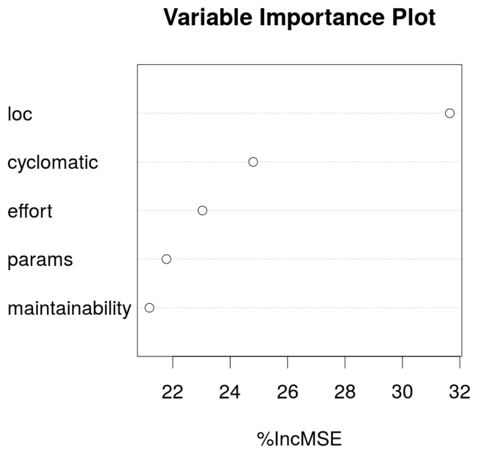


Figure 2-22. Variable Importance plot for the Random Forest Model for Quality for the 520 NPM Packages

2.5.3 Analysis of the NPM Data and Results

We used the same modeling techniques as used in analyzing the mobile applications to examine the interrelationship between the various code complexity measures, extent of usage, and the number of issues. The BN model (Figure 2-19) highlighted that the number of issues is dependent on the number of downloads as well as the number of logical lines of code. The value of R^2 for the BN model was found to be 0.59.

The variable importance plot of the Random Forest model (Figure 2-20) showed the number of logical lines of code to be the most important predictor, followed by the number of downloads over the month before data collection. The mean value of R^2, from performing a 10 times 2 fold cross-validation was 0.64 (sd: 0.04) for this model. We wanted to see how the fit of the model changes if we drop the number of downloads from the list of predictors. This resulted in a significant drop in the value of R^2, which became 0.49 (sd: 0.02) under this condition. This result clearly indicates the importance of the number of downloads in modeling the number of issues even after taking the code complexity metrics into consideration.

Additionally, we decided to implement a usage-adjusted quality measure similar to what we had for the mobile applications and observe how it depends on the code complexity measures. Our quality measure (“Quality") was defined as the number of issues per download, similar to how we defined it while analyzing the mobile applications. The BN model (Figure 2-21) was similar to what we found last time, however, we now observed a new link from “effort” to “Quality”. The fit of the model was found to be somewhat worse, with a R^2 value of 0.41. The Random Forest model gave a R^2 value of 0.53 (sd:0.03) from a 10 times 2 fold cross-validation, and the variable importance plot (Figure 2-22) showed the lines of code to be the most impactful predictor.

2.5.4 Timeline plots for NPM packages

We also presented the timelines for a few well-known NPM packages, showing the comparative trends of the number of issues and our proposed quality measure, defined as the number of issues per download. Since the number of downloads has a large variation, the quality measure also has a high degree of variation. So, we decided to fit a model to the quality variable, and add another line representing the fitted values of the model. We first tried to use a OLS model, but given the apparent non-linearity of the data, later decided to use a Generalized Additive Model (GAM). We did not fine tune this model, because it was only used for demonstrating the trend of the quality measure in the timeline plots.

Since the detailed analysis was performed on single releases of 520 packages, we wanted to verify if the number of downloads is an important predictor for the number of issues for all 4430 packages individually during their lifetime for all releases. We found that out of the 4430 packages, only for 36 packages the p-value of the predictor variable was more than 0.05 before adjusting for the calendar date, and after adjusting for the calendar date, p-value was less than 0.5 (*i.e.* the predictor was deemed significant) for all 4430 packages.

The R^2 values of the fitted models varied between 0 and 0.8637 (median: 0.4982, standard deviation: 0.0618) before adjusting for the calendar date , and between 0.019 and 0.999 (median: 0.9195, standard deviation: 0.0618) after adjusting for the calendar date. So, we can say that the number of daily downloads is a significant and important predictor for the number of issues encountered by the users for most of the packages and the effect is more pronounced when the effect of automated downloads is controlled by calendar date.

Since we just established that the number of issues of an NPM package depends on the number of daily downloads, a similar quality metric of the number of issues per download should be applicable in this situation as well.

To check the quality of different packages, we looked at the minimum and median value of the quality metric for the 4430 packages. We didn’t consider the absolute maximum value, since some packages had zero downloads for a few days, driving the value of the quality metric to infinity. So, we used the 90^{th} quantile value as a proxy for the maximum value. We looked at the packages for which the value of the quality metric was more than 1. The threshold was chosen because we were looking at the packages of really high values of the quality metric, and thus were of poor quality. We found that for 340 packages the 90^{th} quantile value of the quality metric was more than 1, i.e. they had more than 1 issue reported against them per download. The number was 100 and 3 when we looked at the median and minimum values respectively. The three packages for which the total number of issues over the number of daily downloads was more than 1 were '@ngrx/store', '@protobufjs/fetch', and '@protobufjs/inquire'. Overall, we found that the packages for which the value of the quality metric was more than 1 were mostly packages from a big project that were relatively less downloaded, *e.g.* 'babel-plugin-transform-es2015-bLOCk-scoped-functions' from babel project, 'react-scripts' from facebook-react project etc. There were a few other packages that had very few downloads during most of its life-cycle since 2015, but had an increase in popularity later on, and thus were selected in our list of packages. However, since they had very few downloads for a long time, the median or maximum value of the quality metric was more than 1 (*e.g.* 'bubleify'). For illustration, in Figure 2-23 we are showing the histogram of the median value of the quality metric for the 4430 NPM packages, which gives some idea about the overall quality distribution of the packages in the NPM ecosystem. We can see that around 75% of the packages in NPM have a median value of the quality metric less than 0.01, which mean, overall, for around 75% of the NPM packages, less than 1 in 100 regular users ever (since we are looking at the total number of issues) file an issue.

Further inspection showed that the value of the quality variable increases with time for almost half of the packages (2030 out of 4430 packages, 45.8%), unlike what we observed for the mobile applications, where for almost all of the releases of the three softwares, the value of the quality variable decreased with time.

Here we also show the timelines comparing the trend of the quality variable we defined (*i.e.* in this case, number of issues per download), along with a fitted line that was fitted using the Generalized Additive Model (GAM), and the number of issues, for a few selected well-known NPM packages for illustration. The selected NPM packages are quite popular and have a large number of issues reported against them, so so plotted the number of issues in log scale. We can see that for all the four cases, the number of issues keep increasing with a decreasing slope, but the quality measure follows different trends for the four

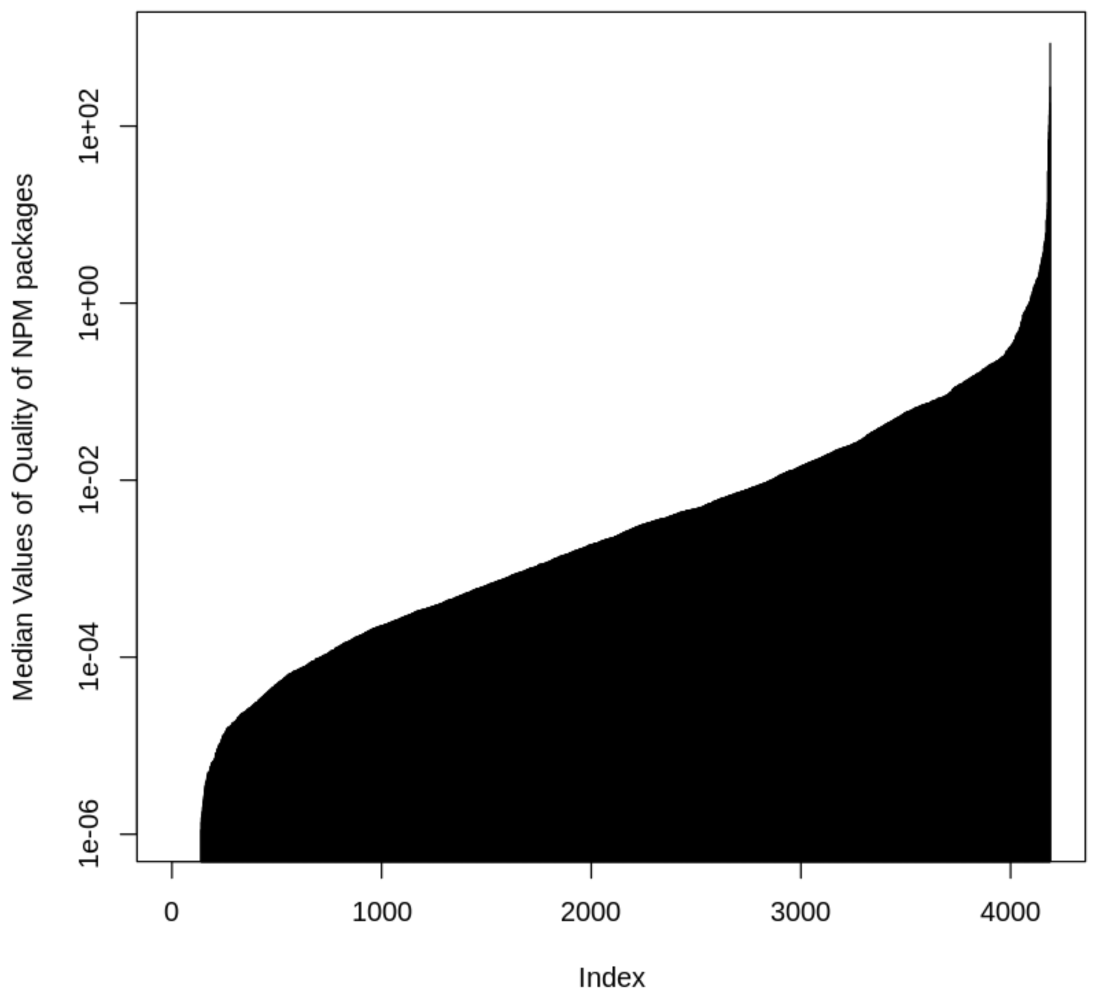


Figure 2-23. Histogram of Median Values of Quality of NPM packages

cases. We see that the quality measure for "angular"(Figure 2-24) and "eslint"(Figure 2-26) have a trend similar to what we saw for the mobile apps, with the value of the quality variable decreasing with time, but "babel"(Figure 2-25) is showing an increase in the value, followed by an initial decrease, while for "ember-cli" (Figure 2-27), the trend is almost constant over time. This result clearly show the necessity of normalizing the number of issues, which is a measure of software faults, by usage parameter like the number of downloads before using it as a measure for software quality.

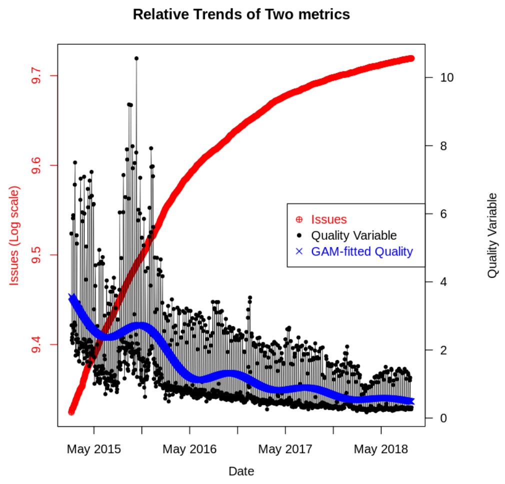


Figure 2-24. Timeline for NPM package: angular

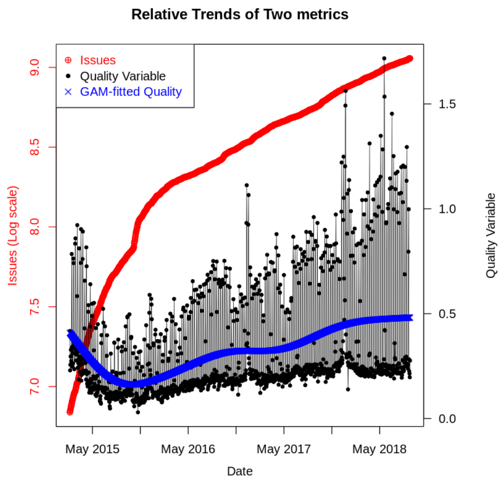


Figure 2-25. Timeline for NPM package: babel

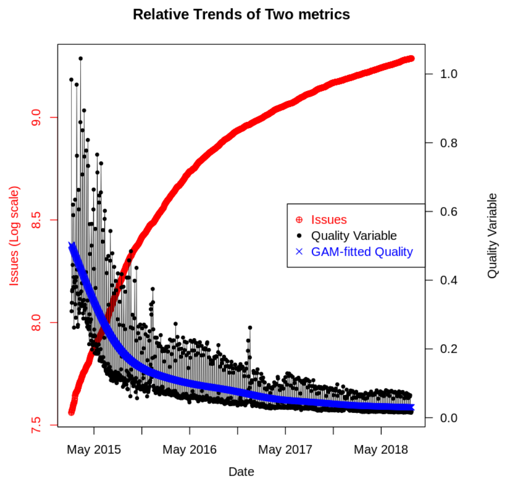


Figure 2-26. Timeline for NPM package: eslint

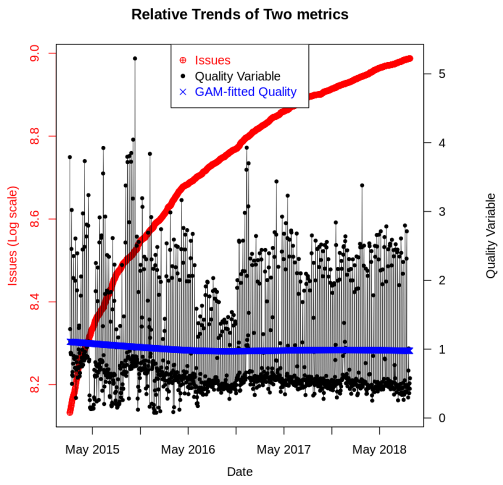


Figure 2-27. Timeline for NPM package: ember-cli

2.6 Discussion

Our analysis makes it evident that the number of users is one of the most important variables in explaining various post-release software failure metrics, as seen in all three of the mobile applications as well as for the NPM packages. The analysis also indicates that, for the mobile applications, more new users for a release would mean more exceptions would be found for the software and, for the GA releases of both the apps analyzed, longer activity for the release (the duration of a release measures how long a release is actively used by users, not the time between two releases, since the releases overlap). This suggests that users may be reluctant to upgrade (or are encouraged to stay) on better -quality releases. For the NPM packages as well, a higher number of downloads indicated a larger number of issues. Our findings are in agreement with findings of [28, 29, 75] that consider post-release defects for a completely different server software system.

The release date also affects no. of exceptions for the mobile applications, as can be observed by looking at the coefficients. It provides some insight on how this software has evolved. Even after adjusting for the effect the number of users have on the number of exceptions, the number of exceptions are increasing with time for the Android app, whereas is decreases for the iOS app. This may indicate that the software, the OS, as well as the hardware could be becoming more complex with time, which is consistent with a rapid growth of functionality and the size of associated code base. The Android app is seeing more crashes due to the variations in the devices and the OS, whereas for iOS, since the devices as well as the OS versions are tightly controlled, the users are seeing less issues, although we have no explicit evidence to support our speculation.

An interesting observation from the model is the *lack of any direct relationship between exceptions and the intensity or frequency of usage*. One possibility is that exceptions happen for specific Android/ iOS version/ Phone combination and the way each user is exercising application's functionality. Users for whom the application crashes must wait for the next release. This would lead to the observed phenomena where only the new users increase the number of crashes, which was observed more clearly from the timeline of crashes as well. The duration an application is used by individual users was found to have a much smaller effect on reported defects than the number of new users in prior work [29, 28, 76] as well. In particular, it was observed that most of the issues happen *soon after deploying the release* and the chances of reporting a defect for a new release drops very rapidly with time after installation.

We found that the exceptions are a result of more new users and the extent of usage does not appear to have a direct effect on the number of new users.

We found that usage of NPM packages, measured by downloads, was a significant predictor for issues even after taking the code complexity metrics into consideration.

We found that among the code complexity metrics, the average per-function count of logical lines of code (*loc*) was the most important predictor for modeling the number of issues, and the effect of the other factors was much less pronounced. From the BN model we found the relationship between the *loc* and the number of issues to be negative, *i.e.* the modules with more average per-function logical lines of code were seeing fewer issues. However, given that this measure is the per-function lines of code, it could indicate that simpler modules with fewer functions are less likely to have issues reported against them.

It is worth mentioning that the most likely reason our models for the mobile applications showed a relatively poor predictive performance is that we did not have any internal measures like the code complexity metrics in those models, and given the number of issues depend on internal as well as external factors (what we saw from the results of our NPM analysis), not having the internal factors affected the predictive performance of those models. Getting code complexity metrics for the closed source mobile applications proved difficult, due to the proprietary nature of the code, and the fact that the development teams worked on multiple releases at the same time further complicated getting the code complexity measures for particular releases. Therefore, we investigated the NPM packages, which are open-source, to verify the impact of usage after taking the code complexity measures into consideration. Since we had both the internal as well as the external factors in the models for the 520 NPM packages, the predictive performance was much better.

From the timeline analysis of the 4430 NPM packages, we observed that the number of downloads is a significant predictor for the number of issues for most of them, and when controlled for the calendar date, which compensated for the variations in the downloads by automated sources, it was a significant predictor for all the 4430 NPM packages. So, a similar quality measure was used for this case as well. We found by looking into this metric that, overall, for around 75% of the NPM packages, less than 1 in 100 regular users ever (since we are looking at the total number of issues) file an issue. However, unlike the three mobile apps, where the value of our quality metric decreases with time for all releases, for the NPM packages the quality metric sometimes increases or remain relatively constant over time (around 45.8% of the time).

Our data, scripts, and more detailed results are available in our GitHub repository: https://github.com/tapjdey/release_qual_model.

Overall, none of the three models indicate that "Usage.Frequency" or "Usage.Intensity" have any effect on the "Quality" variable. We, therefore, suggest that the exceptions per user, or a metric similar to that, can be used as a software development quality metric to objectively compare quality of different releases. While the measure is not very novel or sophisticated (Post-release defect density calculated as a proportion of users who experience an issue within a certain period after installing or upgrading to a new release has been proposed by [77, 75] as a measure of software quality), it is an *actionable* and *easy to use* measure. A more sophisticated approach would require modeling the software failure measures (like exceptions or issues) as a function of software usage, and then use the residuals obtained after fitting the model to objectively compare the qualities of the releases. Such approach may prove to be too complex for a development team to apply.

The wider practical implication of this finding is twofold:

1. Our findings prove that due to the interdependence of usage and the observed number of software failures (like exceptions), *any* quality measure (like number of defects, defect density, mean time between failures) that is dependent on any of these observed number software failures would misclassify some releases being better or worse than others unless the usage aspect is taken into account. The effect naturally would be more pronounced for softwares/releases with a large variation in usage.

2. The results of our findings also suggest that these observed number of software failures do not depend on all aspects of usage, e.g. we found no dependence between usage intensity or frequency and number of observed exceptions. It suggests that to make a quality measure independent of the external factors like usage, we can not just normalize it by any usage measure, e.g. normalizing the number of exceptions by usage intensity or frequency would not make it independent of external factors. It is important to normalize by the right measure to be able to actually make the quality measure independent of usage.

2.7 Related Work

Although software quality has always been a common topic in software engineering [78, 79], most of the studies have focused on pre-release data, primarily due to the developers' concern about finding the appropriate balance between the amount of testing required and the quality of software (e.g. [80, 81]). There have been a number of works on predicting and improving the software quality as well (e.g. [82, 83, 84, 85]). Comparatively, studies about post-deployment quality and dynamics have been less frequent [86, 87]. However, a number of studies have looked at the aspects of software quality metrics, especially the quality perceived by the customers, e.g., [75, 28, 29, 88, 55]. [20] described a general way to measure Software Quality and related metrics for Open Source Software ecosystems. A notable non-academic work involves a study of mobile app monitoring company's (Crittercism) data [89]. The author of the news article found it necessary to normalize crash data by the number of launches. Finally, an empirical investigation between release frequency and quality on Mozilla Firefox has been investigated in [90].

While Bayesian Networks have been used for software defect prediction for decades, the use of BNs for explanatory modeling in empirical software engineering is still not common despite the promise. A case for use of BNs was made by Fenton et.al. [36, 91], while the earliest publications utilizing BNs we could find [92] constructed search of the structure based on the statistical significance of partial correlations in the context of modeling delays in globally distributed development. [93, 94] considered the application of Bayesian networks to prediction of effort, [95, 96, 97] used Bayesian networks to predict defects, and [98] used BN approach for an empirical analysis of faultiness of a software. On the other hand, Bayesian structure learning is a big domain in itself with a wide range of algorithms, but its use in software engineering context is not very common.

Hackbarth et al. [99] found the need to adjust defect counts in their proposed measure of software quality as perceived by customers. We propose a somewhat different measure of quality based on the number of exceptions per user. In general, software quality is a widely researched topic [100, 79, 101] etc., but in our knowledge, this is the first model-based attempt to obtain a usage independent measure of software quality and the first attempt to model exceptions in mobile applications.

The NPM ecosystem is one of the most active and dynamic JavaScript ecosystems and [102] presents its dependency structure and package popularity. [103] studies the dependency, specifically the lag in updating dependencies in various NPM packages while [104] looked into the use of trivial packages as part of package dependencies for different NPM packages. [31] investigated the factors affecting NPM package popularity, and [32] investigated the participation patterns of issue and patch creators.

The advancements proposed in this paper over the published work are focused on two primary areas: (1) study of the relationship between software faults (issues for NPM packages) and usage using **post-release** data in the context of two proprietary mobile applications and 4430 popular NPM packages, and (2) proposing a usage independent exception-based software quality metric based on our models.

2.7.1 Comparison with published results

In this subsection we compare our findings with already reported results that studied other commercial applications. The goal of this subsection is not replicating the earlier studies, but just comparing the findings of our study and those of some earlier studies. We add this section to address the limitation of our dataset having a relatively small sample of data.

Unfortunately, there aren't a lot of studies that looked into the interrelationship between software usage and software faults (defects or crashes).

The number of users for most of the releases we studied are very small, with a median of 7 users per release, although a few releases have more than 16,000 users. On slide 22 of of his presentation [37], Caper Jones reported that the number of defects increase 2 to 3 times for a 10 fold increase in the number of users (from 1 to 10 and 10 to 100) for a software of similar complexity (between 10,000 and 100,000 function points). However, they were looking at the number of defects, and typically the number of exceptions is larger than the number of defects, because one defect could cause crashes for multiple users (or multiple crashes for a single user). The study published in [28] was done for a system with many more users (around 4,000 to 16,000),however, they reported that for a two-fold increase in the number of users the number of Modification Requests (MR tickets) increase around 1.25 times, which is more than what would have been predicted by our model $(1.02 - 1.04)$ for Android apps, but less than what we have (1.6) for the iOS app.

Although we were unable to do a direct comparison to another mobile application, due to different studies looking at different measures, these findings add more context to our result, and indicates the necessity of further studies that publish their datasets to understand the usage-fault relationship in a wider range of applications.

2.8 Limitations

The accuracy of our result is very much dependent on the Google Analytics data. While we do not have reasons to doubt the accuracy of the counts in Google Analytics data, we would have liked to have better definitions of how it determines "New User", "Visit", and, especially, nontrivial to aggregate quantities such as "Visits per User." Also, it is not clear if Google Analytics distorts data in any way (e.g., by applying differential privacy transformations) for low counts in order to protect the privacy of the users. We do not believe it does, but we have not conducted an experiment to validate that.

Furthermore, the mobile applications under consideration were relatively new and it was the first attempt for the team to deploy mobile software. As such, much was not well documented and was rapidly evolving over time. As mentioned earlier, we did not have the official release dates for all releases, so we put the start date of the release as the date on which the first usage was reported. However, we did verify the official dates with this reported date for the releases for which we found the release date, and they were very close, but not always exactly the same. This should not affect the overall result, given the total time scale of more than two years. The release end dates, by their nature, have to be estimated based on user activity, since there is no way to force end user to upgrade Android app. For recent releases, therefore, the end date may be censored by our data collection date, hence the duration for these releases might be underestimated.

Another limitation associated with using these commercial closed-source mobile applications is that we had no control over the release cycle or the variables being measured by Google Analytics. This limited our options for doing the analysis, sometimes severely. We had very few releases for the iOS application, and even the largest dataset of GA releases of the Android application had only 173 releases. We had a limited number of observed variables as well. However, we were unable to obtain any more data on the applications, forcing us to work with the limited data. However, we tried to increase the validity of our study by looking into three sets of releases for two applications, and used three different modeling approaches to study these datasets. The fact that we saw a strong relationship between the number of users and exceptions in all cases has led us to have confidence in the validity of our finding.

It may be possible to collect numerous additional variables that may have an impact on exceptions, for example, the number of changes to the source code made for a release as was done in [28]. Unfortunately, due to the nature of parallel development for multiple releases and products noted in subsection 2.3.1.1, it was virtually impossible to separate the changes that would only affect a specific release on the Android/iOS platform. To further complicate the matter, the mobile applications we studied were commercial in nature, and the source code for these were not available.

Our study of the mobile applications focused on a single set of mobile applications from a specific domain, implemented via a rather complex codebase and is certainly not representative of most mobile applications that tend to be much simpler. Furthermore, mobile applications may not represent other types of software further limiting external validity of the results. However, some aspects that we see in the specific application, such as increasing number of faults with the number of users, has been observed in rather different contexts of large-scale server software. This suggests that the model derived in the study may generalize to other domains as well.

In terms of modeling aspects, there are some limitations related to the different approaches. The RF model was used for 10 times 2 fold cross-validation, and exhibited a rather high value of standard deviation in the R^2 value, likely due to the small sample size.

While creating the BN model we did not cover all possible ways BNs can be applied to gain insight into the system. For example, we did not investigate the possible existence of any hidden node, or make an effort to formally establish the causal relationship between the nodes. We also did not investigate how the properties of one release affect the subsequent releases, nor did we investigate the presence of any feedback loops. Although we used the best methods identified from the simulation study, we did not employ any measures to verify the existence/non-existence of any link that appeared in the averaged bootstrapped model.

In the simulation study, although we covered an extensive set of options, we did not try every possible combination of options for the BN structure search exercise.

We also did not use Markov Random Field analysis, which is another probabilistic graphical modeling approach. The primary reason behind choosing the BN approach was that we found an example where this method was used to successfully recover the underlying network [58]. Moreover, it is possible to interpret a BN model as a causal model, and although we did not use that interpretation in this study, our goal is to eventually establish a causal mechanism of how usage affects the number of exceptions/defects experienced by users, so we wanted to used BN from the start.

Regarding external validity, we analyzed 520 most popular NPM packages, which is less than 0.1% of the total packages in the NPM ecosystem. Even during the timeline study, we only looked at 4430 packages. These packages, however, represent the tiny part of the NPM ecosystem that is widely used, so they constitute a suitable subset for our study.

Although the study of the NPM packages had measures related to code complexity and usage, we didn't look into a lot of other possible variables that could affect the number of issues,*e.g.* the number of dependents a package has. Although some of the issues could come from users of a dependent package, we didn't actively check the origins of the issues to verify that. We also didn't look at the releases of the packages, because of reasons mentioned before. We didn't differentiate between the types of the issues, because we just wanted to see how many times a user decided to file an issue. Overall, this study was not a direct extension of the previous work, rather, it was an extension of the concept and its application in a different domain.

Another approach that could have been taken to make this study more similar to the study of the mobile applications would require us to check whether or not an issue filed for a package has a crash report. However, such an approach would come with different caveats, e.g. a crash could result from the limitation of the package, but it could also result from some bug or compatibility issue in the web browser, or even the OS. Due to these limitations, we did not investigate this in this study, though, it is an interesting question we would like to address in future.

2.9 Summary: Addressing RT1

The goal of this project was to address the first research topic mentioned in subsection 1.2.1: *"Investigating if the perceived quality of a software depend on its usage even after accounting for the code complexity measures, and, if we can design a easy-to-use and usage independent measure for software quality."*

The results of this project established that the usage of a software, measured by the number of new users for the Avaya mobile applications and by the number of downloads for the NPM packages, has a strong influence over the software failure measures, measured by the number of exceptions for the Avaya mobile applications and by the number of issues for the NPM packages, which can be used as proxies for the quality of software from the users' perspective. The study of the 520 NPM packages revealed that the effect remains noticeable even after taking the code complexity measures into account. Therefore, counting exceptions, or using any other quality measure dependent on an observed number exceptions, or any other software failure metric (like number of defects or issues), therefore, will not accurately measure the quality of software development process but, instead, it would strongly depend on the extent of use. In order to produce a measure that the development team can use to understand and improve quality of their software development process, we proposed to normalize the observed exceptions by usage, specifically by number of users or any related measure if it is not available. Notably, a similar normalization was previously proposed in the context of post-release defects that also exhibited strong positive correlation with the number of users. By studying software from two different domains, we have also shown that the concept is extendable to different types of software. The study revealed that even a less accurate measure of usage like downloads, which, for NPM packages, is a mix of downloads by human users and automated sources, is an important predictor for the number of issues reported, which again is a weakly similar measure to the number of crashes or bugs. So, our approach can be applied to any situation similar to the ones we studied, even when only proxy measures for usage and crashes/ bugs are available. The study also revealed the importance of taking software usage into account even in the presence of code complexity measures.

Finally, much more work is needed to gather additional empirical evidence of how software behaves post-deployment. It is important to note that Google Analytics data is available only for application developers, so while each project has the ability to see their app's performance, they can not see data for software created by other organizations. This can be addressed by a) projects sharing theirs post-deployment data (we have not seen examples of that); or b) publishing findings based on such data in cases such as ours, where the data itself would be impossible to release publicly since it involves numerous, often enterprise, customers who may not agree.

CHAPTER 3

ESTIMATING THE EFFECT OF INDIRECT USAGE ON SOFTWARE POPULARITY

3.1 Overview

This chapter addresses the interrelationship between the popularity of a software and its *indirect* usage, measured by its dependencies on other software modules, and also the modules that depend on it. Thereby, it tries to address the second research topic posed in subsection 1.2.1 of Chapter 1: “Modeling the effect of indirect usage (e.g. dependent packages) of a software on its popularity.” Some of the research about this topic was published in a conference paper [31] in the PROMISE conference in 2018.

3.2 Background

A software with a large user base tends to have a large support group, meaning getting more help in terms of detecting and, possibly, resolving bugs, more questions and answers on Stack Overflow, more and more readily available tutorials, discussions, and blogs, and, potentially, more volunteers to work on related projects [47]. Therefore, understanding the underlying mechanism that drives the popularity of a software is crucial for understanding how the sociotechnical systems represented by software ecosystems thrive.

Some contemporary methods of software deployment make use of the easy online distribution. Instead of making monolithic softwares, many open-source software ecosystems like NPM, R-CRAN, Python-Pip etc. have adapted the registries that specify dependencies among components needed for deployment, in which one module depends on one or more other modules for functioning. Such dependencies could be runtime, development, or of some other type (for this study we focused only on runtime dependencies). Due to the nature of deployment, this dependency structure can play a crucial role in determining the popularity of software packages, because when one package is installed by a user, all of its runtime dependencies (referred to simply as “dependencies” henceforth) are also downloaded and installed automatically (unless they are cached). Therefore, the number, and more importantly the popularity of packages that depend on a particular package should strongly influence its popularity.

Although the interdependence between the popularity of a software and its dependents may appear as rather obvious to the software development community, we found no studies that investigated if it holds or considered by how much the component's popularity may be explained by the number and popularity of its dependents. Therefore, we propose the following research questions in this study:

RQ1: Do the number and popularity of the dependents of a software have any significant impact on its popularity? What is the proportion of variance explained by these variables?

RQ2: Do the number and downloads of dependencies and dependents of a project help predict its change in popularity?

We investigated the node package manager (NPM) ecosystem to answer our questions because of the large size of this ecosystem and the availability of data. NPM is a package manager of JavaScript packages, and is one of the largest OSS communities at present, with over 1 million packages. Moreover, NPM tracks the number of downloads for the packages, which is one of the most direct and easily available measures of popularity [31], and makes the data publicly available. However, the size of the ecosystem also makes an analysis of the whole ecosystem extremely challenging. We, therefore, decided to limit the scope of our study to packages with more than 100,000 monthly downloads since January, 2018 (4804 in total or less than 0.5%).

Corresponding to RQ1, we found that the number as well as the popularity of dependents of a package have very strong influence over its popularity, and only the number of dependents and the download count of the most popular dependent package could explain between 73% and 100% of the variation in popularity for the 4804 packages under consideration.

With respect to RQ2, our findings suggest that both the number and the popularity (measured by the number of downloads) of the dependencies are some of the most important predictors for the change in downloads for NPM packages. These models can help interpret the download counts and reasons that affect the usage of the package. Our findings also suggest that the download number can be affected (or manipulated) through dependencies and frequent updates (probably leading to cache misses).

3.3 Related Work

The topic of software popularity hasn't seen much attention, possibly due to the lack of reliable popularity measures. Stars for GitHub projects were used to identify the factors impacting the popularity of GitHub projects [48]. The relationship between popularity of mobile apps and their code repository was studied in a number of papers, e.g. [105, 106, 107, 108, 109], where the ratings of the mobile apps were used as a measure of popularity. Other studies that investigated the popularity looked at the popularity of Python projects on GitHub [110], the relationship between the folder structure of a GitHub project and its popularity [111], effect of popularity on software crashes [26] etc.

The NPM ecosystem is one of the most active and dynamic JavaScript ecosystems and [102] presents its dependency structure and package popularity. [103] studies the dependency, specifically the lag in updating dependencies in various NPM packages while [104] looked into the use of trivial packages as part of package dependencies for different NPM packages, [32] investigated the effort contribution and demand patterns by contributors in the NPM ecosystem.

Zerouali et. al.[112] reported 9 different popularity measures that have been used by various studies for the NPM ecosystem, including the number of direct and transitive runtime dependents of a package, the number of downloads, npm-stars [1], and the number of stars, forks, pull requests, and subscribers of the GitHub repositories of these packages. Dey et. al. [31] used the number of downloads, the same measure used in this paper, as the measure of popularity (although they used the number of monthly downloads and we're using the number of daily downloads) and showed that the number of dependencies and dependent packages have an impact on predicting whether the popularity of a package will increase or decrease *in future*. However, they did not establish the relationship between the popularity of a package and its number of dependents and the popularity of those dependents *in the same time period*, nor did they report the proportions of variance explained, which is our goal in this paper.

3.4 Methodology

In this section we describe the data collection process and the analysis steps used to answer the research question we posed.

[1]https://docs.npmjs.com/cli/star

3.4.1 Data Collection

Keeping our research questions in mind, we needed two types of data for our analysis. First of all, we needed the download counts of all packages of interest, and we also needed to find the dependents and dependencies for all packages in question.

We started by collecting the number of monthly downloads for all packages in the NPM ecosystem from the `npms.io` website, using the API provided [2]. Our data was collected based on a snapshot of the NPM ecosystem taken on 2018-08-31, and we collected information about 782,359 NPM packages. After filtering for our criteria that the NPM package must have more than 100,000 monthly downloads (since January, 2018), we were left with 4804 different NPM packages.

Finding the list of dependents needs some extra calculation because the `package.json` file of a package only lists the upstream dependencies of a package, so we needed to go through the `package.json` files of every package in the NPM ecosystem to find the list of dependents for a given package. To obtain the metadata information for every package in NPM, we wrote a “follower" script [3]. The output contained the metadata information for all releases of all packages in NPM. From this we constructed the reverse mapping from a package to all packages dependent on it, and the timeline of the number of dependents for the 4804 packages.

3.4.2 Analysis Method

3.4.2.1 Methodology for Addressing RQ1:

For addressing RQ1, We obtained the daily download counts of the 4804 packages we analyzed, and the 376,389 packages that are dependent on at least one of these 4804 packages, from 2015-03-01 to 2018-08-31. We used the API provided by NPM for this purpose [4]. After looking into the collected data, we noticed that for a few days, the number of downloads for all the packages was reported as 0, which could be because of the downtime of the servers that keep track of the download counts, or some database error, or something else. To keep our data consistent, we removed the entries for those dates, which are: *2018-05-29 to 2018-06-01, 2018-08-06, and 2018-08-28*. After initial data

[2] https://api.npms.io/v2/package/[package-name]
[3] https://github.com/npm/registry/blob/master/docs/follower.md
[4] https://github.com/npm/registry/blob/master/docs/download-counts.md

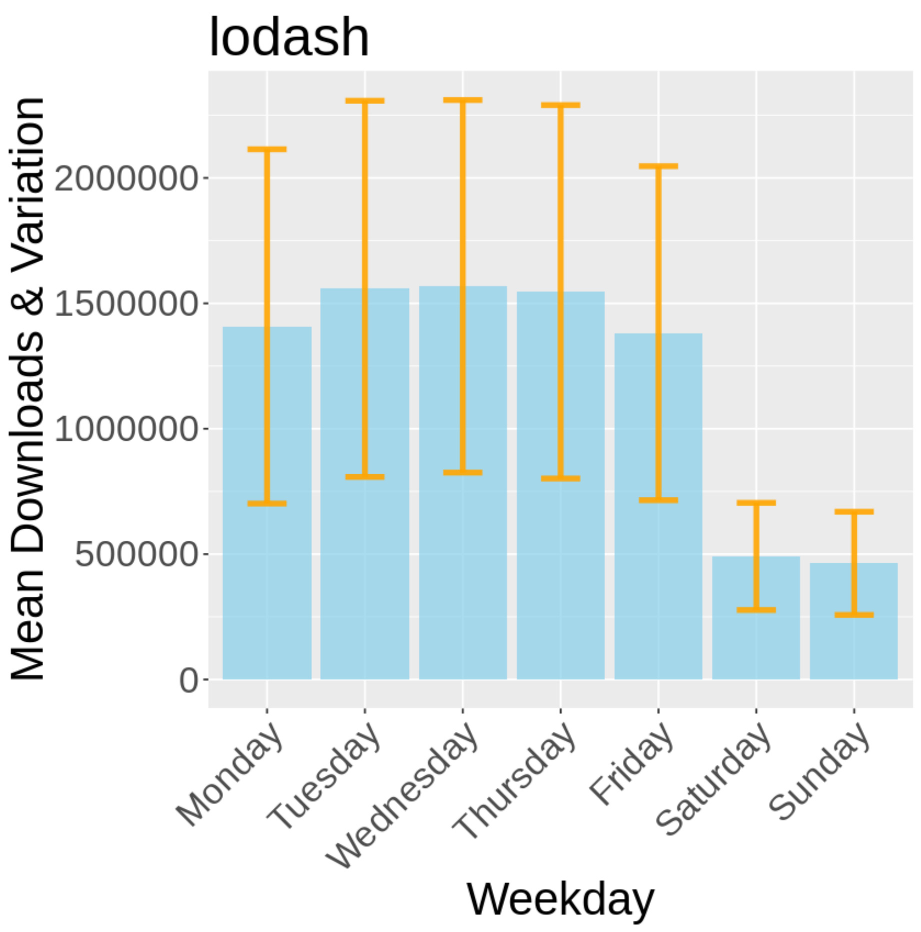


Figure 3-1. Mean Number of Downloads (with standard deviation) on different days of the week Between 2015-03-01 and 2018-08-31 for NPM package “lodash”

exploration, we noticed a few things that impacted our variable selection and choice of modeling methodology:

1: The number of downloads for the NPM packages were drastically different between weekdays and weekends, with weekends showing significantly lower number of downloads and even a different rate of increase in the number of downloads over the years, an example of which can be seen from Figure 3-1 for the NPM package “lodash”, and the trend is very similar for all NPM packages. Therefore, we decided to use two different models, one for the weekdays, another for the weekends, to answer our research question.

2: Although the packages under consideration had a lot of dependent packages (between 28 and 62545, with a median of 283 for all the packages over the time of analysis), most of the dependents had very few downloads. Therefore, we decided to investigate the impact of popularity of the most popular dependents of the 4804 packages under consideration on their popularity, in addition to investigating the impact of popularity of all dependents of a package on its popularity. Upon preliminary investigation with data on 10 randomly selected packages, we found that the downloads of the most popular dependent package as a predictor yields pretty good goodness of fit, and when we looked at models with the downloads of top "N" most popular dependent packages as a predictor, the goodness of fit value saturated at $N = 10$. Therefore, we decided to test the impact of the most popular dependent, and the 10 most popular dependents on the popularity of these packages.

3: While the number of downloads and the number of dependents increased linearly for a few NPM packages, for most of the packages the growth rate was non-linear. Therefore, instead of using a simpler linear regression model, we decided to use Generalized Additive Models (GAM) as our methodology of choice (we used the implementation of GAM in the "mgcv" package in R [113]). We decided not to use methodologies like Support Vector Machine (SVM) or Random Forest regression because they are not suitable for testing the validity of a hypothesis.

Based on our observations and corresponding decisions, we formed 4 different predictors for answering RQ1: (1) the number of dependents of a package on a particular day, (2) the total number of downloads of the dependents on that day, (3) the number of downloads of the most popular dependent on that day, and (4) the total number of downloads of the 10 most popular dependents on that day; and used these predictors for estimating the number of downloads of that package on that particular day. However, since these predictors, especially the pair of predictors 2 & 3, 2 & 4, and 3 & 4 were observed to have high values of correlation for most of the packages, we decided to use 7 different models to properly test the significance of each of the predictors and estimate the proportion of variance explained by the relationships, while mitigating the multicollinearity effect: 4 models each with one of the 4 predictors, and 3 other models with predictor pairs 1 & 2, 1 & 3, and 1 & 4. Also, as mentioned above, we constructed 2 different sets of models for the weekdays and weekends. So, finally, we fitted $7 * 2 * 4804 = 67,256$ different GAM models for answering our first research question.

3.4.2.2 Methodology for Addressing RQ2:

To understand the effect of dependencies and dependents of package on its popularity, we decided to look at the monthly values for its downloads, because the daily and weekly downloads for most packages exhibit large variations, which, presumably, are random in nature, and we do not have sufficient history to predict yearly downloads for most packages. We collected snapshots of the statistics on all relevant packages roughly every 2 weeks between 01-December-2017 and 15-March-2018, and used the data from a snapshot to predict the number of downloads for the packages over the next month. To ensure as little overlap as possible, we did not use the data from a snapshot to predict the number of downloads in the next snapshot, which is 2 weeks apart, but looked at the one after that, which is roughly 1 month apart. Also, since the actual number of downloads varies drastically among packages, we use the logarithm of the ratio of downloads in next month and the downloads of the previous month as our response variable. To focus on the effect of the supply chain aspects we decided to consider only packages with at least one dependent and one dependency.

Since the response variable for our study is the logarithm of the ratio of downloads in next month and the downloads of the previous month, in order to maintain consistency, we removed all other download variables from our dataset. Keeping in mind the goal of our study, we designed 12 other variables to examine the influence of upstream and downstream dependencies on our response variable. We constructed the list of one-level (immediate) upstream and downstream dependencies as well as the recursive (all, as mentioned earlier) upstream and downstream dependencies for a package. For each of these four, we calculated the total number of dependencies, the total number of downloads for those dependencies (for the current month), and the average (we used median instead of mean because having a highly popular package in the list would skew the mean substantially) number of downloads for the dependencies. These variables were named using the *convention*: “(one/recursive).(up/down)stream.(dl.avg/dl/count)", where “dl" indicates the total number of downloads, “dl.avg" indicates the average number of downloads, and “count" indicates the number of dependencies. Due to the skewed nature of all the predictor variables, we took logs of all the variables for our analysis.

We performed the analysis of the data in two steps: First, we ran Linear Regression (LR) individually on each of the 6 snapshots, with and without the 12 dependency related variables, and compared the results. In this step we only performed model fitting, and no prediction. In the second step, we combined data from all the 6 snapshots, added a “Date” variable to account for the seasonality component, and fitted a Random Forest (RF) model and performed prediction (70% of the data was used for training and 30% was used for testing). We performed a 10-fold cross-validation using Random Forest by recursively reducing the number of predictors (less important predictors removed first) in each step to obtain the optimal number of predictors by trading model complexity and cross-validation error. Then we verified how many and which of the 12 variables we introduced are in the optimal set of predictors so obtained, and the model performance with these predictors. We used the adjusted R^2 value as our performance metric, since we are trying to predict the ratio of downloads, which is a continuous variable. As a side analysis, we also fitted the Random Forest model on the combined data with a binary response variable that indicates whether the number of downloads will go up in the next month, i.e. if the ratio will be more than 1 or not. For this step, we calculated the AUC under the ROC curve as well as the sensitivity and specificity reported by the model, with the same 70-30 ratio for training and testing sets. The analysis was performed in R.

3.5 Results

In this section we report some of the general trends we observed about the NPM ecosystem to put the rest of findings in context and address the research question we posed.

3.5.1 Observed Trends in the NPM Ecosystem

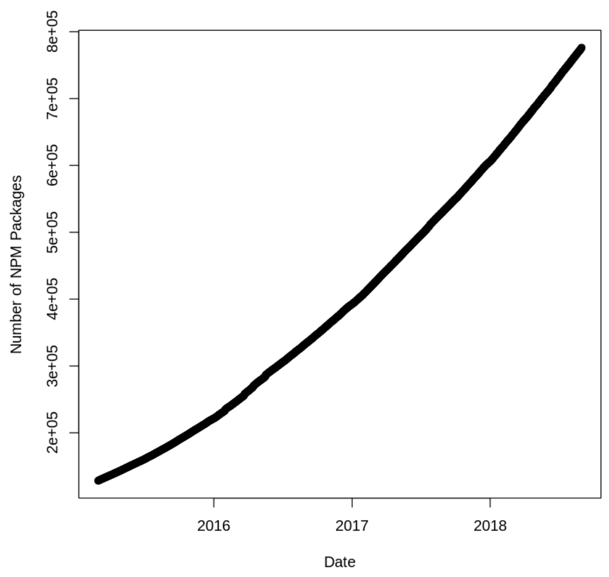


Figure 3-2. Number of NPM projects over time

We observed that the NPM ecosystem showed a steady growth during our analysis period, as can be seen from Figure 3-2 which shows how the number of packages in NPM increased during our analysis period. The number of users of the NPM packages have also increased accordingly, with about 4000 new users on an average day[5]. So it is unsurprising that the number of dependents of the packages under consideration, as well as the number of downloads of the packages have increased with time.

Another factor that likely contributed to the increase in the number of dependents is that "*[p]eople very seldom rip packages out of software once they're installed*" [47]. This is also supported by our analysis, which found 639,256 instances when a dependency was added, and only 78,890 instances when a dependency was removed, i.e. we found that addition of a dependency to be 8 times more common than deletion of one. Furthermore, we also noticed that 48.64% of the dependencies were added in patch releases, 23.28% in minor version releases, and 28.08% in major version releases. Conversely, 18.93% of the deletion of a dependency happened during major releases, 24.26% during minor releases, and 56.81% during patch releases. So, around half of all the additions and deletions of dependencies happened during patch releases of the NPM packages.

3.5.2 Addressing RQ1: To what extent do the popularity of the dependents of a software affect its popularity?

Table 3-1. Performance of different predictors for the 4804 NPM packages

Model predictor	**Weekday Model**		**Weekend Model**	
	R^2 **[min value - max value (median, standard deviation)]**	**p-value (Worst Case)**	R^2 **[min value - max value (median, standard deviation)]**	**p-value (Worst Case)**
No. of dependents	0.43 - 0.97 (median: 0.9, sd: 0.06)	<1e-10	0.45 - 0.98 (median: 0.93, sd: 0.07)	<1e-10
Total download of dependents	0.54 - 1.00 (median: 0.98, sd: 0.06)	<1e-10	0.16 - 1.00 (median: 0.96, sd: 0.11)	<1e-10
Download of the most popular dependent	0.06 - 1.00 (median: 0.97, sd: 0.13)	<1e-10	0.03 - 1.00 (median: 0.95, sd: 0.19)	<1e-3
Download of 10 most popular dependents	0.34 - 1.00 (median: 0.98, sd: 0.07)	<1e-10	0.08 - 1.00 (median: 0.96, sd: 0.12)	<1e-7

As mentioned in Section 3.4, we fitted 67,256 different GAM models to verify whether the popularity of an NPM package depends on its number of dependents and their popularity, and estimate the proportion of variance explained. Table 3-1 shows how each of the four predictors performed in explaining the popularity of the 4804 packages under consideration. We noticed that individually, all of the predictors were significant in explaining the number of downloads (popularity) of an NPM package even in the worst case in terms of the p-values reported by the GAM models. Now, Young et. al. [114] reported that p-values for the non-linear terms are underestimated in the GAM models, however, S.N. Wood, the author of the "mgcv" package in R which includes the implementation

[5]https://twitter.com/seldo/status/880271676675547136

Table 3-2. Result of models with the pairs of predictors for the 4804 NPM Packages

Model Predictors	R^2 **value for the weekday model [min value - max value (median, standard deviation)]**	R^2 **value for the weekend model [min value - max value (median, standard deviation)]**
No. of dependents + Total Downloads of dependents	0.82 - 1.00 (median: 0.98, sd: 0.03)	0.73 - 1.00 (median: 0.97, sd: 0.05)
No. of dependents + Total Downloads of Top 10 popular dependents	0.82 - 1.00 (median: 0.98, sd: 0.03)	0.73 - 1.00 (median: 0.97, sd: 0.05)
No. of dependents + Downloads of the most popular dependent	0.73 - 1.00 (median: 0.97, sd: 0.04)	0.73 - 1.00 (median: 0.96, sd: 0.05)

of the GAM model we used, later stated that [115] (more accurate) hypothesis tests were implemented " in the function `summary.gam` of the R package mgcv from version 1.7-14 onwards."

Therefore, we can conclude from our tests that the number of dependents as well as their popularity have significant effect on the popularity of an NPM package.

Looking at Table 3-1, we notice that while each of the predictors achieved a very high value of adjusted R^2 both for the best case as well as for the median case, the worst case values are significantly lower, both for the weekday models and weekend models. However, when we looked at the performances of the models that used the pairs of predictors (as stated in Section 3.4), we noticed that the worst case adjusted R^2 values improved significantly, as can be observed from Table 3-2. Furthermore, the models with pairs of predictors had a higher median value of R^2 and the standard deviation in the values of R^2 was much lower. Another interesting fact that we can observe from Table 3-2 is that the model with the number of downloads of the most popular dependent of a package and the number of dependents as predictors performed almost as good as the model with the number of downloads of all dependents, and the model that had the number of downloads of just the top 10 most popular packages did just as good. This phenomenon shows the prevalence of extreme disparity of popularity among the dependents of a package. It was seen that, for a number of packages(like "restore-cursor"), a large portion (up to 99%) of downloads of all dependents came from just the most popular dependent package, though for some packages, like "lodash", the portion was as low as 7%.

The number of dependents and the number of downloads of the most popular dependent can explain the popularity of an NPM package with a high degree of confidence, with R^2 values ranging between 0.73 and 1, and considering the downloads of the top 10 most dependents can further improve the performance in some cases, while no further improvement is achieved by considering the downloads by the rest of the dependents.

3.5.3 Addressing RQ2: Can the popularity of the dependencies and dependents of a package predict its change in popularity?

The linear regression models had adjusted R^2 values ranging between 0.02331 and 0.17300 (median: 0.05301, standard deviation: 0.05373) for the models with the 12 variables we introduced, and between 0.00957 and 0.15640 (median: 0.02008, standard deviation: 0.05646) without those variables. Moreover, the adjusted R^2 value with our extra variables was seen to be better for every snapshot. The Variance Inflation Factor(VIF) for the predictors never exceeded 5, so we do not have a serious multicollinearity problem in this analysis. This leads us to believe the variables we added increase the explanatory power of the model. The set of significant variables were seen to be different between different snapshots. The variables representing the counts, downloads, and average downloads of the recursive downstream dependencies of a package were seen to be significant in five out of the six snapshots, and number of releases and commits in last one year were significant in four. The coefficients for total recursive downstream downloads and number of commits were positive; for number of releases it was positive in two, negative in two, and for the rest the coefficients were always negative. Since we used logs of all the variables, the signs of the coefficients indicate the signs of the powers of the variables in direct relationship with the response variable.

The analysis performed in this step also revealed that the total number of downloads of all NPM packages was very different across the snapshots, indicating the presence of a strong seasonality component, therefore, we added the date of data collection as a seasonality variable when we aggregated the data from different snapshots for further analysis.

We ran a cross-validation exercise using Random Forest by recursively reducing the number of predictors. The exercise indicates the optimal number of predictors to be **19**, from our original set of 27 predictors. The variable importance plot, ordered by percent increase in mean-squared error, of all variables is shown in Figure 3-3, the variables listed

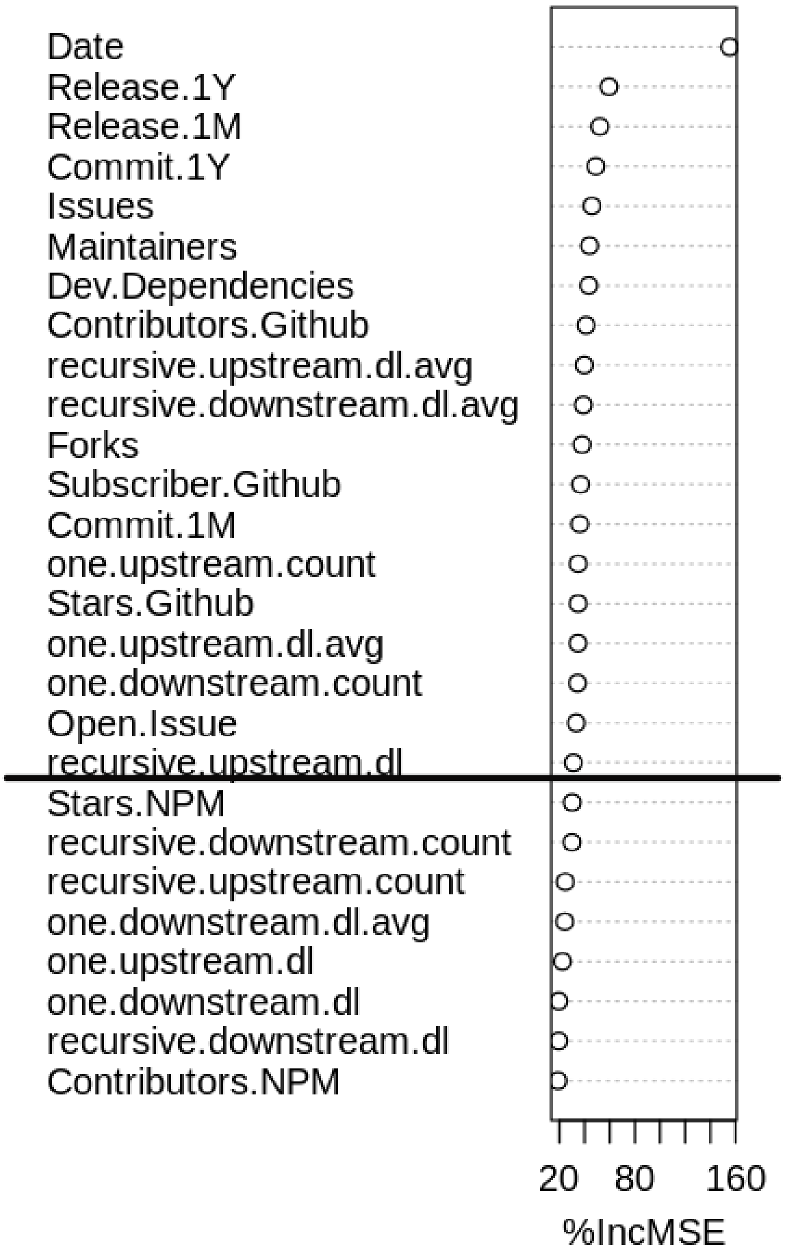


Figure 3-3. Variable Importance plot

above the straight line are the top 19 predictors. It can be seen that 6 of the 12 variables we introduced are in the list of top 19 predictors. The top predictors include the date of data collection, as a seasonality component, which is the most important predictor by far, along with the number of releases and commits in a year and a month, number of issues (total and open), number of forks, stars, contributors, and subscribers to the GitHub repository of the package, number of development dependencies, with the number of immediate upstream and downstream dependencies, the average popularity (average downloads) of all upstream and downstream dependencies, overall popularity (total downloads) of upstream packages, and average popularity of immediate upstream dependencies.

Finally, we ran a 10 fold cross-validation using the 19 predictors deemed important from the analysis in the last step. The resultant adjusted R^2 varied between 0.380 and 0.496 (median: 0.416, standard deviation: 0.031).

The set of variables that were shown as important from the Random Forest analysis are different from the ones that came out as significant from the LR analysis. One of the reasons for that is likely due to the fact that when we fitted the LR models, we did not perform any prediction, so the variables deemed significant from that analysis are listed based on their explanatory power, but the importance of the variables from the RF analysis are listed according to their predictive power, and these two criteria are not the same [64].

When we used the Random Forest model (with all 27 variables) for predicting if the ratio will be more than one (i.e. if the downloads will increase), the value of AUC under the ROC curve was 0.73, and values of sensitivity and specificity were 0.66 and 0.56 respectively. When the model was fitted without the 12 variables we introduced, the values of AUC under the ROC curve, sensitivity, and specificity were 0.65, 0.59, and 0.53 respectively.

The count and popularity of upstream and downstream dependencies of a package are significant both in terms of explanatory power (as revealed by LR analysis) and predictive power (as revealed by RF analysis). It is also evident that among the dependency variables, the recursive ones are equally or more important, highlighting the value of the supply chain perspective.

3.6 Limitations

In terms of internal validity of the result, we modeled the popularity of only a tiny fraction of projects in the NPM ecosystem. However, since these are the most popular packages in NPM, we believe that modeling the popularity of other packages is of lesser interest for understanding the mechanism that drives the popularity of the most downloaded packages.

In terms of external validity, we studied only one software ecosystem, so the results may not generalize for other ecosystems. However, given that a number of software ecosystems employ a distribution process that is in many ways similar to the NPM registry, there is a possibility that insights from this study might transfer to those ecosystems as well.

3.7 Summary: Addressing RT2

The goal of this project was to address the second research topic mentioned in subsection 1.2.1: *"Modeling the effect of indirect usage (e.g. dependent packages) of a software on its popularity."*

The results of this project have shown that the number of dependents and their popularity can almost entirely explain the popularity of a software package, and they can be important predictors for predicting whether the popularity of a package will increase or decrease. The effects of recursive (transitive) dependencies and dependents were also found to equally important, showing the benefit of adopting a supply chain view of the software ecosystems. We also found that addition of dependencies is 8 times more common than the deletion of one, suggesting that the ecosystems are becoming increasingly interconnected and complex, so the supply chain perspective would be essential for answering many of the upcoming questions about the software ecosystems.

Further studies are needed to understand why the other factors have marginal association with package popularity and if this means that packages used more frequently by end-users aren't as popular. Looking at different types of packages (build tools, test tools, general components etc.) and identifying the dependency networks and effects of change in the network for them can also give valuable insights.

CHAPTER 4

EXPLORING THE RELATIONSHIP BETWEEN A SOFTWARE PACKAGE AND THE DEVELOPERS WHO CREATE ISSUES AGAINST IT

4.1 Overview

This chapter explores what fraction of the developers who crates issues against a software are related to the package through the software supply chain. Thereby, it tries to address the third research topic posed in subsection 1.2.1 of Chapter 1: "Exploring if the issue contributors of OSS projects depend on the project through dependency supply chain, and if the contributors who aren't dependent on the package through the supply chain have a different characteristics than the contributors who are." Most of the research about this topic was published in a paper [32] that was published in the PROMISE conference in 2019.

4.2 Background

Open Source Software is characterized by the fact that the source code is publicly available and can be modified and reused with limited restrictions by the public. This has led to the creation of user communities that contribute regularly to the development process [11, 116], primarily through reporting and, in many cases, fixing bugs [9]. Reported bugs, in effect, create a demand for effort needed to address them, and, it has been extensively documented (see, e.g. [117]) that large numbers of low-quality issues may overwhelm the projects. Some users also provide patches with their issues (pull requests or PRs) which should require, at least in principle, much less effort to address, and can be regarded as a contribution of effort to the project (the effort spent by the user to create the patch). We refer to PRs when talking about effort contribution and issues without patches when talking about the demand of effort in the further discussion. Since our data is collected from GitHub, which treats pull requests as issues, we follow the same terminology in our paper, i.e. when we talk about "issues", we refer to both issues with and without patches. Many of the contributors, who create these issues and patches, are potential developers and/or developers of their own projects. When we refer to “users” in this paper, we actually are referring to this population of user-developers.

Software ecosystems, by their nature, enable creativity and productivity by letting developers not to write software from scratch but only focus on incremental improvements that depend on other modules in the ecosystem for bulk of the functionality. This results in a complex supply chain of dependencies within the ecosystem. The supply chain of a user consists of the direct as well as transitive dependencies on repositories to which they maintain. While some studies found that contributing and demanding effort is more complicated when it crosses project boundaries in direct and transitive dependencies [118, 119], it is not clear how prevalent such contribution and demand is at the ecosystem level.

This question is closely associated with the concept of visibility [20] within a supply chain, which refers to how far a user can “see” in the supply chain beyond their direct upstream dependencies, i.e. if they are aware of the transitive dependencies of the projects they are using. This is an important question since a lack of visibility in an ecosystem is detrimental to the users’ capacity to contribute, leading to a limitation in user innovation, potential licensing conflicts due to a transitive dependency using a different license, exposing users to higher risk due to the user not being aware of bugs upstream that can be used to instigate a supply chain attack [12] and various other types of risks (see, e.g. [120, 121, 122, 123]), and various other problems. Visibility within a supply chain is not easy to measure, however, the number of cross-project issues and PRs is a good proxy. An ecosystem with greater visibility would allow its users to be able to contribute to their transitive dependencies more frequently. Therefore, by measuring where the issues and PRs are concentrated in the supply chain, we can get a good sense about the level of visibility within the ecosystem. Thus, we present our first research question as:
RQ1: Where in the supply chain do the contribution of effort and demands for effort occur?

We are also interested in discovering if we can identify different groups of users based on their participation patterns, i.e. in which layer of their respective supply chains they contribute effort or demand effort from. The answer to this question is important to identify

[1]https://it.slashdot.org/story/19/06/08/1940204/how-npm-stopped-a-malicious-upstream-code-update-from-stealing-cryptocurrency

[2]https://www.bleepingcomputer.com/news/security/somebody-tried-to-hide-a-backdoor-in-a-popular-javascript-npm-package/

and characterize different sub-communities of users within the ecosystem. So, the second research question we are addressing in this paper is:

RQ2: Can we identify different groups among the users based on their participation patterns?

It has been previously observed that the so called one-time-contributors [124, 125] might have different motivation and behave differently from more involved participants. We, therefore, would like to understand if such distinctions apply in large software ecosystems, i.e. if more active developers contribute different proportion of their effort to upstream projects than casual users. We identify the more prolific users as those who have submitted at least 10 issues to the ecosystem under consideration (we are not counting the issues submitted to other ecosystems). The number 10 is somewhat arbitrary, but, given that 75% of the users in our sample submit 3 or fewer issues, it only includes individuals representing less than ten percent of all users.

RQ3: Do the answers of RQ1 and RQ2 change if we consider only the more prolific users?

Finally, commercial entities tend to participate in FLOSS in ways that are distinct from the way volunteer or independent developers participate [126], which stem from a number of facts, like differences in motivation, interest, urgency, and expertise. Therefore, we would like to understand if such distinctions apply in large software ecosystems, and, in turn, if the participation patterns can be used to predict if a user has a commercial affiliation. GitHub user profiles have the option of declaring if a user works for a company, and, it can be argued that more serious users take their time to populate their profiles accurately. However, the Git version control system extends far beyond GitHub, and a model that can identify the commercial affiliation of a user by looking into their participation (issue and PR creation) patterns would be useful in classifying the types of users in platforms that do not have this option. Thus, our last research question is:

RQ4: Can we use the participation patterns of users to predict their commercial affiliation?

We chose node package manager (NPM) to answer our research questions because of the size of the ecosystem, availability of data, and the large number of its users who work for a company. NPM is a package manager of JavaScript packages, and is one of the largest OSS communities at present, with over 932,000 different packages (Apr, 2019) and millions of users (estimated 4 million in 2016 [127], and about 4000 new users on an average day[3]. NPM is used heavily by companies. According to the NPM website[4], all 500 of the Fortune 500 companies use NPM, and they claim that: " *Every company with a website uses npm, from small development shops to the largest enterprises in the world.*" Given the heavy industry use of NPM, a good number of the users who contribute to it are likely to have a commercial affiliation, which should give us a more balanced dataset to answer **RQ4**. However, most packages in NPM are not widely used and have limited or no issues or PRs. We, therefore, focused on 4433 NPM packages with over 10,000 monthly downloads since January, 2018, that also had an active GitHub repository with at least 1 issue. All issues ever filed against these packages were obtained using the GitHub API (pull-requests are treated as issues by the GitHub API) resulting in 1,376,946 issues and PRs, out of which 541,715 (39%) were pull-requests. We also retrieved information for the 272,142 still active users (some users who filed issues had deleted their accounts and had their id replaced the special GitHub id "ghost").

Our primary findings are: (1) Users are more likely to contribute issues and PRs to their direct dependencies, but a number of issues were created for packages outside a user's supply chain, and very few cross-project issues and PRs were observed. (2) Three different user groups were observed based on the users' effort demand patterns, those who are likely to create issues to their direct dependencies, those who are likely to create issues to packages none of their public repositories depend on, and a small group of users who are likely to create cross-project issues. Based on the effort contribution patterns we observed two major groups, similar to the first two groups observed based on the effort demand pattern. (3) We see that more prolific users are even more likely to contribute to their direct dependencies and much less likely to contribute to packages outside their respective supply chains. (4) We were able to identify the company affiliations of the users with 70% accuracy (95% Confidence Interval between 69.9% and 70.54%) with their contribution patterns as predictors using a tuned Random Forest model, with the value of AUC under the ROC curve being 0.68.

4.3 Related Work

The NPM ecosystem is one of the most active and dynamic JavaScript ecosystems and [102] presents its dependency structure and package popularity. Studies on NPM

[3]https://twitter.com/seldo/status/880271676675547136
[4]https://www.npmjs.com/

have mostly focused on its dependency networks [128], its effect on popularity of NPM packages [31], and problems associated with library migration [129].

As a part of our study we look at the dependencies of the JavaScript projects in GitHub, and the different NPM packages. However, we look not only at the direct dependencies, but also into the transitive dependencies of the packages, i.e. dependencies of dependencies of the packages. A number of studies looked into the handling of dependencies of NPM ecosystem in particular. E.g., [103] conduct an empirical study on the lag in updating a package in conjunction to its dependencies in NPM and its effect, while [130] conduct an comparative study of dependency handling by NPM, R-CRAN, and RubyGems ecosystems, and compare the different strategies used by the three in handling dependency updates.

Our first research question looked into the aspect of issue reporting and the prevalence of cross-project issues in NPM ecosystem. The number of observed issues and PRs is directly dependent on the amount of usage, as reported in [26]. [131] showed that failures in upstream packages brought more and more troubles to the downstream projects. An approach to identify Cross-System-Bug-Fixings in FreeBSD and OpenBSD kernels was proposed by [132]. Other studies in this topic explored how the downstream developers find the root causes and coordinate with upstream developers to fix the problems [133], the workarounds employed by downstream developers when faced with a bug in an upstream project [134], and the question of how to automate the fix of a bug introduced by a third party library upgrade [134]. Unlike these studies, we focus on both effort demand and supply and employ a much larger data-set of projects.

One of our research questions center around predicting users with a company affiliation based on the differences in the types of contributions. Just because a user is affiliated to a company doesn't necessarily imply that they use the NPM packages for their job applications, but it may increase that likelihood. Our belief in this assumption is bolstered by the result of the 2018 Node.js User Survey Report[5], which found that: *"A majority* (of users of NPM packages) *are developers (as opposed to dev managers), in small (<100 employees) companies, with 5+ years of professional development experience."* Given the typical user base, we believe it is a fair assumption that a significant number of users who have disclosed that they have a company affiliation, actually use these packages as a part of their day job and not as a hobby.

[5]https://nodejs.org/en/user-survey-report/

FLOSS development started with the goal of emphasizing the freedom of computer users[6]. Although initially the commercial software development community steered clear of open source software, its benefits, as discussed in studies like [135], soon led them into using and supporting open source software development. A plethora of studies looked into the scenario of commercial adoption of open source software, e.g. [136, 137] to name a few. Currently, the interaction between open source software and different software companies is much stronger and closer, with many companies actively supporting open source development, and using different open source software in a daily basis. Although a number of studies looked into the benefits of using open source software by a company(e.g. [116]), and the result of commercial involvement [138, 126] by studying different project level metrics like sustainability, developer inflow and retention etc., to our knowledge no study has looked into the difference in types of contribution of individual commercial and non-commercial users on a large scale software ecosystem like NPM and used it for predicting if a user has commercial affiliation.

4.4 Methodology

In this section, we discuss some terminologies we used in this study and discuss the analysis method we followed.

4.4.1 Terminologies

Our research questions look into the packages where a user creates issues and PRs, and at which level of the user's supply chain these packages belong to, and we define some terminologies describing these levels for the ease of referring to these levels.

The NPM packages that a user(developer) contributes to directly are referred as *level 0* packages for that user, i.e. only users who have committed to an NPM package directly, and not through a pull request, can have *level 0* packages. Arguably, these user-developers are part of the core team of that NPM package, since they have direct write access to that repository.

The direct dependencies of all repositories a user has ever committed to (we utilize a recent version of WoC data [30] to collect information from all repositories, including projects that are not registered in NPM) are called *level 1* packages for that user.

[6]https://www.gnu.org/philosophy/floss-and-foss.en.html

Furthermore, *level 1* packages also includes originating packages that the said user has forked.

The direct and transitive dependencies of the *level 1* packages are classified as *level 2+* packages of the user. Contributions to *level 2+* packages can be regarded as cross-project contributions by the said user, since these are transitive dependencies for them. The reason we referred to level 2 or higher packages by aggregating them into *level 2+* is that the number of reported issues dropped drastically starting from level 2. Moreover, since any issue reported at level 2 onward would be qualified as a cross-project issue, such aggregation seemed reasonable.

The remaining packages in NPM ecosystem are *level X* packages for that user, since these include all the packages none of the public repositories the user has committed to depend on even transitively. For obvious reasons, we could only observe the publicly visible repositories the user-developer committed to. These packages are the ones that are outside a user's supply chain, but for the sake of consistency and ease of referring, we call them *level X* packages.

The issues and PRs created by a user for a package which belongs to one of these levels of the supply chain for that user are regarded as the issues and PRs created for that level by that user.

4.4.2 Analysis Method

The data collection was done using Python, and the analysis was performed using R.

We started by collecting the necessary data, which was used to create our final dataset. The data collection and data processing steps are described in detail in Section 4.5.

Python scripts were used to create the data files necessary for analysis. We carefully tabulated the number of issues and PRs created for each level of the supply chains of the users to address our first research questions.

To answer RQ2, we decided to calculate the marginal probabilities of each user creating an issue and a PR to each level in their respective supply chains. However, we observed only around 1 in 3 users create a PR, and looking into the two probabilities together would have automatically put 2/3rds of the users in one group and the rest in other. So, we decided to look only at the probabilities of users creating issues (at different levels in their respective supply chains) when looking at all users, and look at the probabilities of users creating PRs (at different levels in their respective supply chains) only for the subset of users who have created at least one pull request.

We used the fuzzy c-means clustering algorithm [139] for answering RQ2. We decided to use this instead of the more commonly used k-means or hierarchical clustering algorithm because we suspected, and later observed, that there is a lot of overlap in our data, and k-means doesn't work well with such data; as for hierarchical clustering, given we have 272,142 users in our dataset, calculating the distance matrix needed to construct the clusters proved very difficult due to the computational resources required. The fuzzy c-means algorithm assigns membership probabilities to each data point instead of assigning them to clusters directly, which gives the best results for the type of data we have. We used the fuzzy c-means implementation in the *e1071* R package, and for visualizing the clusters we used the "clusplot" function in the *cluster* R package.

We used Random Forest model (*randomForest* package) for training our predictive model (RQ4), since it is one of the best performing models. The model parameters ("ntree" and "mtry") were tuned using functions from *caret* and *e1071* packages.

4.5 Data Description

In this section, we describe the data collection and data processing steps, focusing on the design choices that were made along the way.

4.5.1 Data Collection

Keeping our research questions in mind, we needed the following types of data:

1. The list of NPM packages that satisfy our criteria of having more than 10,000 downloads per month and a GitHub repository with at least one issue.
2. Link to GitHub repositories of these packages for collecting the issues.
3. List of all issues and issue creators of these packages.
4. Detailed information on the issue creators to know if they disclose their company affiliation.
5. List of all commits made by these users, and the list of GitHub repositories where they made those commits.
6. List of source repositories of the forked repositories the users may have committed to.

7. List of all dependencies (NPM packages) of the GitHub repositories the users committed to.
8. List of dependencies of all NPM packages for creating the transitive list of dependencies for the repositories the users committed to.

The data for item (1) was collected from the `npms.io` website, using the API provided [7]. The associated GitHub repository URL (item 2) and the list of dependencies of the NPM packages (used for item 8) were collected from their metadata information, which was obtained by using a "follower" script, as described in NPM's GitHub repository [8]. After filtering for our criteria that the NPM package must have more than 10,000 monthly downloads (since January, 2018), a functional link to its GitHub repository, and at least one issue, we were left with 4433 different NPM packages.

The list of all issues for the packages (item 3) was obtained using the GitHub API for issues[9], using the `state=all` flag. We ended up with 1,376,946 issues (until January, 2019, when the data was collected) for the 4433 packages. It is worth mentioning here that sometimes more than one NPM package can have the same associated GitHub repository, e.g. all TypeScript NPM packages (starting with "@types/", like @types/jasmine, @types/q, @types/selenium-webdriver etc.) refer to GitHub repository "DefinitelyTyped/DefinitelyTyped". To avoid double-counting and further confusion, we saved the issues keying on the repository instead of the package name, though we also saved the list of packages associated with a repository. We found that there are 3797 unique repositories associated with these 4433 packages.

Then we extracted the list of all users who created these issues and obtained detailed information on them (item 4) using the GitHub API[10]. We found that there were 272,142 users still active (as of March, 2019, when the data was collected) out of 280,835 users who had created issues for the NPM packages under consideration.

For obtaining information on items (5) and (6), we used the GHTorrent database [140] available in the Google Cloud platform[11] (we used the `ghtorrent-bq:ght_2018_04_01` database), and extracted the relevant information using Google BigQuery.

[7]https://api.npms.io/v2/package/[package-name]
[8]https://github.com/npm/registry/blob/master/docs/follower.md
[9]https://developer.github.com/v3/issues/
[10]https://developer.github.com/v3/users/
[11]http://ghtorrent.org/gcloud.html

To get a list of all projects a user ever committed to (item 5), we extracted the list of commits made by a user and got the list of the repositories where those commits were made, finally getting the list of all repositories the user committed to. We found that the 272,142 users committed to 6,676,089 projects in total, and it had a very skewed distribution in terms of the number of projects a user committed to. Note that these projects don't have to be JavaScript projects, since we obtained this information from all Git data [30]. Upon further analysis, it was found that 5,898,782 of them had a `package.json` file, so we classified them as JavaScript projects, and used them for further analysis.

For getting the sources of the forked repositories the users might have committed to (item 6), we used the *projects* table in the GHTorrent database, which has a field named "forked_from", and performed a recursive search (since project A can be forked from B, and B can be forked from C etc.) to get the list of all sources.

For the data in item (7), we extracted information for all GitHub repositories that has a `package.json` file and extracted the dependency information from that. We also found that some repositories use another file named `lerna.json` to list their dependencies. So, we extracted dependency information from this file as well where it was available.

There were cases where the users directly committed to a package repository. Those were treated as special cases and handled using a map of package name and package URL constructed previously.

The transitive dependency map of item (8) was constructed by doing a recursive search using the dependency information collected for the packages. We listed the direct dependencies of a package as level 1 dependencies of that package, the dependencies of the packages in level 1 as level 2 dependencies of that package, and so on. It is worth mentioning that if a package A, for example, was found to be dependent on a package B directly, as well as through another package C (A depends on C, C depends on B), we took the lower number, i.e. B was still listed as level 1 dependency of A. Moreover, although forks are not dependencies of a project in the same way other dependencies work, we decided to add the sources of the forked repositories as level 1 dependencies for ease of representation. However, from level 2 onward, we only have packages in the list of dependencies, which includes the dependencies of the source repositories of the forked ones.

Table 4-1. Final List of Variables in the Dataset

User login $	No. of projects the user committed to	No. of repos that are forks of other repos	No. of NPM packages committed to
No. of direct dependencies of all the user's packages	No. of transitive dependencies of all the user's packages	Total no. of issues created by the user	Total no. of PRs created by the user
No. of issues created for level 0 packages	No. of issues created for level 1 packages	No. of issues created for level 2+ packages	No. of issues created for level X packages
No. of PRs created for level 0 packages	No. of PRs created for level 1 packages	No. of PRs created for level 2+ packages	No. of PRs created for level X packages
Total no. of packages for which a issue was created	Total no. of packages for which a PR was created	No. of level 0 packages for which an issue was created	No. of level 0 packages for which a PR was created
No. of level 1 packages for which an issue was created	No. of level 1 packages for which a PR was created	No. of level 2+ packages for which an issue was created	No. of level 2+ packages for which a PR was created
No. of level X packages for which an issue was created	No. of level X packages for which a PR was created	Total no. of issues that are not pull requests	No. of non-pull-request issues created for level 0 packages
No. of non-pull-request issues created for level 1 packages	No. of non-pull-request issues created for level 2+ packages	No. of non-pull-request issues created for level X packages	If the user has a company affiliation $

4.5.2 Data Processing

The raw data was processed to create a usable dataset for analysis. For each user, we first extracted the list of repositories they contributed to and then constructed the list of packages they transitively depend on. The transitive (level 2+) dependencies for a user was calculated using the transitive list of dependency data (item (8) above). Then we extracted the packages the user had raised issues for, and observed if that package belongs to level 0, 1, 2+, or X for that user.

We noticed that the user id that created issues to the most number of packages was found to be "ghost", which is of little surprise, and it was removed from subsequent analysis. The second and third positions were occupied by two bots associated with the automated dependency management website/service Greenkeeper[12], both of which raised issues for more than 400 different packages, and created pull-requests for 98% of those packages,

[12]https://greenkeeper.io/

and 92% of the issues raised by these two bots were pull-requests. We further noticed that bots tend to create a lot more issues and PRs compared to human users. So, we decided to remove the users that we could identify as bots, because bots are much more prolific by design, and could skew the distributions significantly. We were able to identify 35 bots which were removed from further analysis.

The variables in our final dataset are listed in Table 4-1. Each entry in the table is the observation for one user. All variables, except User login and whether the user has company affiliation (marked by $ in Table 4-1), are numerical in nature.

4.6 Results

Table 4-2. Distribution of Issues created by Users for different levels in their respective supply chains
Numbers on the right show the values for users with 10 or more issues, pertaining to RQ3

	Fraction of issues created for Level 0	Fraction of issues created for Level 1	Fraction of issues created for Level 2+	Fraction of issues created for Level X
All users who created an issue	0.027 | 0.039	**0.532 | 0.688**	0.039 | 0.039	0.402 | 0.234
Users who created issue for level 0	0.139 | 0.127	**0.761 | 0.778**	0.028 | 0.028	0.071 | 0.067
Users who created issue for level 1	0.033 | 0.041	**0.760 | 0.772**	0.039 | 0.039	0.168 | 0.148
Users who created issue for level 2+	0.031 | 0.034	**0.679 | 0.728**	0.116 | 0.077	0.174 | 0.160
Users who created issue for level X	0.019 | 0.029	**0.456 | 0.652**	0.035 | 0.042	0.490 | 0.278

Table 4-3. Distribution of Pull Requests (PRs) created by Users for different levels in their respective supply chains
Numbers on the right show the values for users with 10 or more issues, pertaining to RQ3

	Fraction of PRs created for Level 0	Fraction of PRs created for Level 1	Fraction of PRs created for Level 2+	Fraction of PRs created for Level X
All users who created a PR	0.048 | 0.056	**0.772 | 0.810**	0.020 | 0.015	0.160 | 0.119
Users who created PR for level 0	0.171 | 0.155	**0.791 | 0.809**	0.009 | 0.009	0.029 | 0.027
Users who created PR for level 1	0.047 | 0.057	**0.884 | 0.881**	0.014 | 0.014	0.054 | 0.049
Users who created PR for level 2+	0.042 | 0.044	**0.843 | 0.868**	0.055 | 0.033	0.06 | 0.055
Users who created PR for level X	0.034 | 0.038	**0.727 | 0.794**	0.018 | 0.016	0.222 | 0.152

In this section we discuss our findings and answer the different Research Questions we had, staring with some general statistics about the data. Since our RQ3 is asking the same questions as our RQ1 and RQ2, but with a different condition, we present the answer of RQ3 together with the answers of RQ1 and RQ2.

4.6.1 General Statistics about the Data

Here we discuss some general statistics, which, in spite of not being directly related to our research questions, can give us some insight into the data and the NPM ecosystem in general.

To recap, our study focused on 4433 NPM packages (3797 unique GitHub repositories) with more than 10,000 monthly downloads since January, 2018. We collected 1,376,946 issues created for these projects, including 541,715 pull-requests, which were created by 280,835 users, out of whom 272,142 were active at time of data collection.

A few interesting statistics about the data are reported below:

- We found that 219,945, or around 81% of the total users had committed to at least one public repository in GitHub.
- 84,813 (31%) users have a disclosed company affiliation, but they created almost 57% of the pull-requests, and around 42% of the issues.
- 87,653 (32%) users had created at least one pull request, or, 68% of the users have created issues but never submitted a pull request.
- 38,080 (14%) users have never submitted any issue without a patch, i.e. all the issues they submitted were PRs.
- 4585 (1.7%) users in our user base had committed to at least one NPM package directly, so were likely part of the core team of an NPM package.
- 139,917 (51%) users committed only issue, i.e. just over half of the users who committed at least one issue were "one-time-contributors", and they create around 11% of the total no. of issues, and around 4.6% of the total no. of PRs.
- 215,584 (79%) users committed at least one issue, and 31,330 (12%) of the users committed at least one PR to a package not in their supply chain (level X).

- 21,144 (8%) and 4643 (1.7%) users committed at least one issue and at least one PR respectively, to a transitive dependency package. i.e. they submitted cross-project issues and cross-project pull requests respectively.
- 89,149 (33%) and 62,262 (23%) users committed at least one issue and at least one PR respectively, to a direct dependency package.
- Only 19,376 users had created more than 10 issues (corresponding to our condition in RQ3), which consists of roughly 7% of the entire user population, but they create around 60% of the total issues and 75% of the total PRs.
- All of the numerical variables listed in Table 4-1 have extremely skewed distribution.

Previous studies of contribution patterns reported a layered structure of a core team, bug fixers, and bug reporters for individual projects (see, e.g. [141, 142]). We see a similar distribution of the users, with 1.7% of the users likely to be part of the core team of some package, 32%, who provide patches, could be thought of as bug fixers, and the rest, which consists of the majority of the user base, are issue reporters. This shows the premise of the onion model is valid at the ecosystem level as well.

4.6.2 Where in the supply chain are the contribution of effort and demands for effort concentrated? (RQ1), and, does the distribution change for the more prolific users? (RQ3)

To answer this question we looked at the number of issues and PRs created by each user at different levels of their supply chain, as defined in Section 4.4.1. The results of the finding are reported in Tables 4-2 and 4-3, where the distribution of issues and PRs created by users for different levels in their respective supply chains are reported in terms of the fraction of issues and PRs reported at each level. The values on the left side are the fractions for all users under consideration, and the values on the right side are the fractions for the more prolific users.

We observe from Table 4-2 that, when considering all users, most of the issues (53.2%) are reported for the direct dependencies of the users, followed by issues created (40.2%) for packages on which none of the users' public repositories depend on. The fraction of cross-project issues is pretty small (3.9%), and so is the number of issues created for level 0 packages(2.7%). When looking at the more prolific users, the fraction of issues created for level 1 packages increases further, and the fraction of issues created for level X packages gets reduced, while the other two remain almost similar. This indicates they are more likely to create issues for their direct dependencies and less likely to create issues for packages none of their public repositories depend on, while their likelihood of creating issues for level 0 and level 2+ packages remain similar to the likelihood for all users.

We also decided to look at the conditional distributions of issues, that are created by users who have created at least one issue to a particular level in their respective supply chains. We noticed that the fraction of issues created for level 1 packages is significantly increased when we focus only on the users who have created at least one issue for a level 0 or level 1 package. Looking at the users who created at least one cross-project issue, the fraction of issues created for level 1 packages is still increased, but by a lesser amount, while the fraction is reduced when we focus on users who created at least one issue for a level X package. This indicates the users who create issues for a level X package are likely different from the rest, which we investigate further while answering RQ2.

While looking at the distribution of pull requests (Table 4-3), we see a trend very similar to the one we saw for the issues, with the fraction of PRs created for level 1 being even larger under all condition, and the fraction being smaller for level X packages. The fraction under the different conditions also follow a trend similar to what saw for issues.

In summary, looking at the distribution of issues, we notice that most of the issues are created for the users' direct dependency packages, but a number of issues are also created for packages on which none of the users' public repositories depend on even transitively, which wasn't something we expected. As for pull requests, we see more of them being created for level 1 packages, but again, a number of PRs are being created for the level X packages. When looking at the more prolific users, we see even more issues and PRs being created for level 1 packages, and less issues/ PRs being created for level X packages, but the fraction of issues/PRs being created for level 0 or level 2+ packages don't change by much. Also, we observed very few cross-project issues, and even fewer cross-project PRs under all conditions.

4.6.3 Can we identify different groups among the users based on their participation patterns? (RQ2), and, does the distribution change when we look at the more prolific users? (RQ3)

We discussed the analysis method used to answer this research question in Section 4.4.2. We ran the fuzzy c-means clustering algorithm 4 times, once with the marginal probabilities of all users creating an issue (*Case I*), and once with the marginal probabilities of users, who have created at least one PR, creating a PR (*Case II*) at different levels of their respective supply chains. Then we repeated the same with the

Table 4-4. No. of members and Probabilities of creating issues at different levels for the cluster centers for Cases I and III

	Case I			Case III		
	Cluster 1	Cluster 2	Cluster 3	Cluster 1	Cluster 2	Cluster 3
No. of members	78047 (29%)	8520 (3%)	185575 (68%)	5612 (29%)	8932 (46%)	4832 (25%)
Probability of creating issue in level 0	0.002	0.01	0.001	0.02	0.01	0.004
Probability of creating issue in level 1	**0.952**	0.03	0.007	0.53	**0.89**	0.050
Probability of creating issue in level 2+	0.006	**0.92**	0.001	0.13	0.02	0.012
Probability of creating issue in level X	0.040	0.04	**0.991**	0.32	0.08	**0.934**

Table 4-5. No. of members and Probabilities of creating PRs at different levels for the cluster centers for Cases II and IV

	Case II		Case IV	
	Cluster 1	Cluster 2	Cluster 1	Cluster 2
No. of members	58826 (67%)	28827 (33%)	12842 (80%)	3127 (20%)
Probability of creating PR in level 0	0.007	0.01	0.01	0.04
Probability of creating PR in level 1	**0.974**	0.02	**0.95**	0.14
Probability of creating PR in level 2+	0.007	0.02	0.01	0.04
Probability of creating PR in level X	0.012	**0.95**	0.03	**0.78**

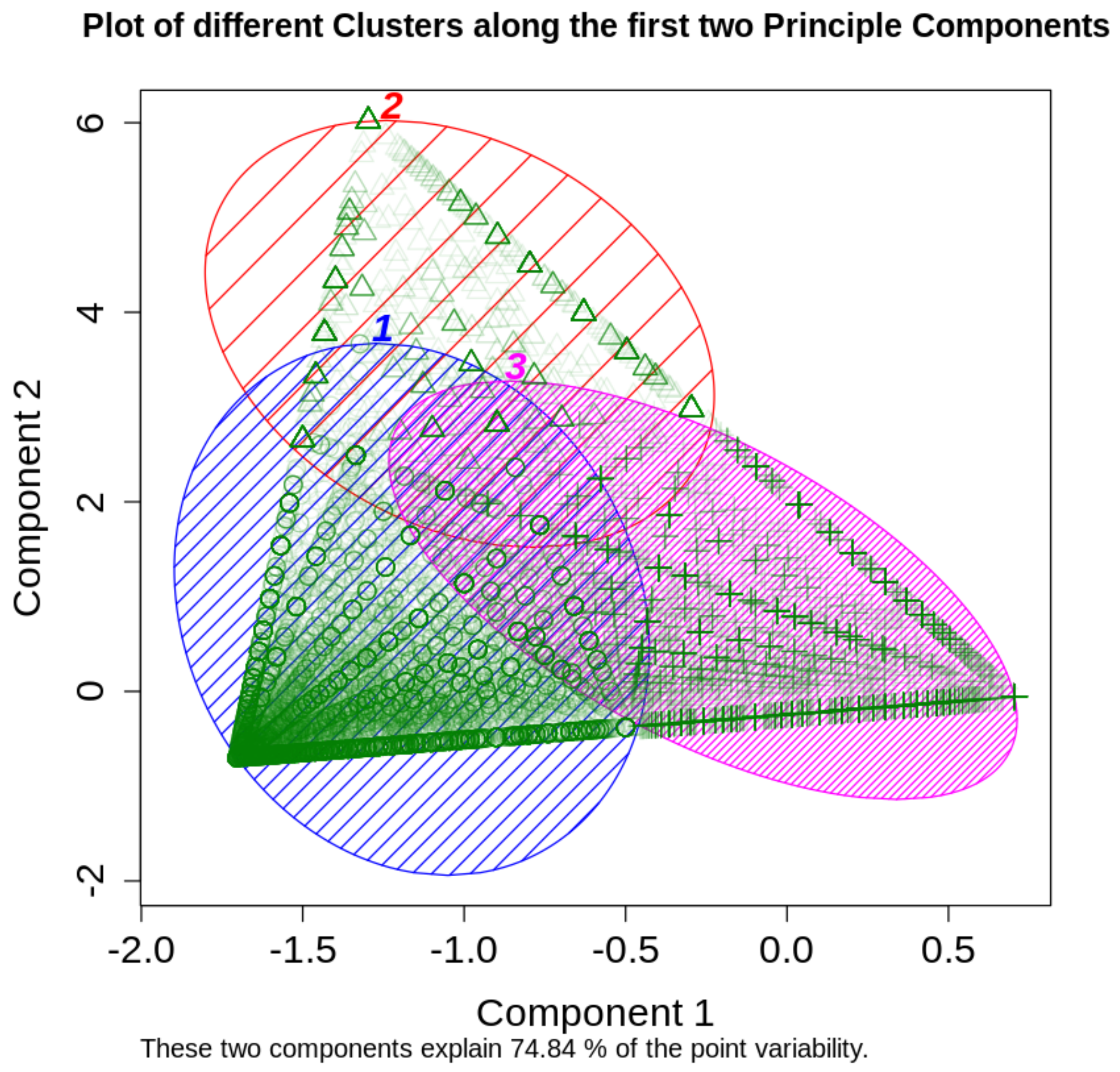


Figure 4-1. Visual Representation of the 3 clusters for Case I

users who have created 10 or more issues *(Cases III and IV)*. For the sake of brevity we only show the visual representation of the clusters created for all users' probabilities of creating issues (Case I). The others are available in our GitHub repository: *https://github.com/tapjdey/NPM_user_analysis*, along with our code and other results.

Looking at the result of clustering, **we noticed 3 different clusters for Cases I and III,** however, **for Cases II and IV, we found two major clusters.** We show a visual representation of the clusters created for Case I in Figure 4-1, where the data points (in *green*) are plotted along the first two principle components, and the three clusters are shown as the three shaded regions. Since the first two components explain around 75% of the data, we assume this is a fairly accurate representation.

We show the number and percentage of data points in each cluster, along with the cluster centers for Cases I and III in Table 4-4, and for Cases II and IV in Table 4-5. Since we used the probabilities of users creating issues and PRs as our data source, the cluster centers indicate at which level of their respective supply chains the users in that cluster are more likely to contribute issues and PRs to.

Looking at Table 4-4, we notice that for Case I, more than 2/3rds of all the users (cluster 3) belong to the group who are very likely to create issues for packages in level X, around 29% of the users (cluster 1) are avid contributors to their direct dependencies (level 1), and a small group of users (3%, cluster 2) also exists who contribute heavily to their transitive dependencies (level 2+), i.e. they are very likely to create cross-project issues. For the more prolific users (Case III), we see a slightly different picture. Although we again see a group of users who contribute heavily to their level X projects (cluster 3), the percentage of the users is reduced to only 25%, while the population of users who contribute heavily to level 1 projects (cluster 2) now consist of around half (46%) of the population. Once again, we see a group of users (around 29%) who are much more likely than the overall population average to contribute cross-project issues (cluster 1), but these users also contribute a lot of level 1 issues, and some level X issues as well.

From Table 4-5, we notice that 2/3rds of the users (cluster 1, Case II)) who have created at least one pull request are very likely to create them for their direct dependencies, while the rest (cluster 2) are more likely to create issues for their level X dependency packages. Looking into the more prolific users (Case IV), we notice that the percentage of users who are likely to create PRs to level 1 packages (cluster 1) is increased to 80%, while the other 20% (cluster 2) are more likely to create PRs to level X packages, but they also create a number of PRs for level 1 packages, and are more likely to create PRs for level 0 and 2+ packages.

We examined the amount of activities of different users belonging to different clusters and found that the users who commit more to their direct dependencies are more active, creating more issues and PRs, and committing to more repositories, while the users more likely to commit to level X packages show very little activity and many of them have company affiliations. The users who are likely to create cross-project issues tend to have a large number of transitive dependencies, and create very few PRs. All of these differences were significant, which was verified using the Kolmogorov-Smirnov test.

In summary, we see three different groups of users based on which level of their respective supply chains they create issues for. While a large number of users are likely to create issues for level X packages, a group consisting of a good number of users are more likely to create issues for level 1 packages, and a small group of users also exists who are likely to create cross-project issues. In terms of creating PRs, we see two major group of users: 2/3rds of the users are more likely to create PRs to level 1 packages, while the rest are more likely to create PRs for level X packages. Looking into the more prolific users, we again see three groups of users based on their issue creation patterns, but the percentage of users who create issues for level X packages is reduced, and the fraction of users who create issues for level 1 packages is increased. As for the users who created at least one PR and 10 or more issues, the fraction of users belonging to the group who are very likely to create PRs to level 1 increase even further, while the rest of the users form a group who are more likely to contribute PRs to level X packages.

4.6.4 Using participation patterns of users to identify their company affiliation (RQ4)

To answer this question, we used Random Forest modeling technique, as mentioned in Section 4.4.2. Our dataset had the predictors at listed in Table 4-1. We dropped the predictor "User.login", and were left with 30 predictors and our response variable was the binary variable representing if the user had a company affiliation. To obtain the optimal number of predictors we used the "rfcv" function from the *randomForest* R package, which shows the cross-validated prediction performance of models with sequentially reduced number of predictors (ranked by variable importance) via a nested cross-validation procedure. Looking at the output of this function, we decided to use 7 predictors for our final model.

First, we created a Random Forest model with all the predictors, and selected the top 7 predictors by looking at the variable importance plot. To calculate the performance of the model, we decided to use 70% of the data, selected randomly, as our training set, and the other 30% as our test set. Then, to optimize our model, we decided to tune the model parameters, viz. "mtry", the number of variables randomly sampled as candidates at each split, and "ntree", the number of trees to grow. We used the "train" function from the *caret* package in R for performing a grid search on the training data to find the optimal values

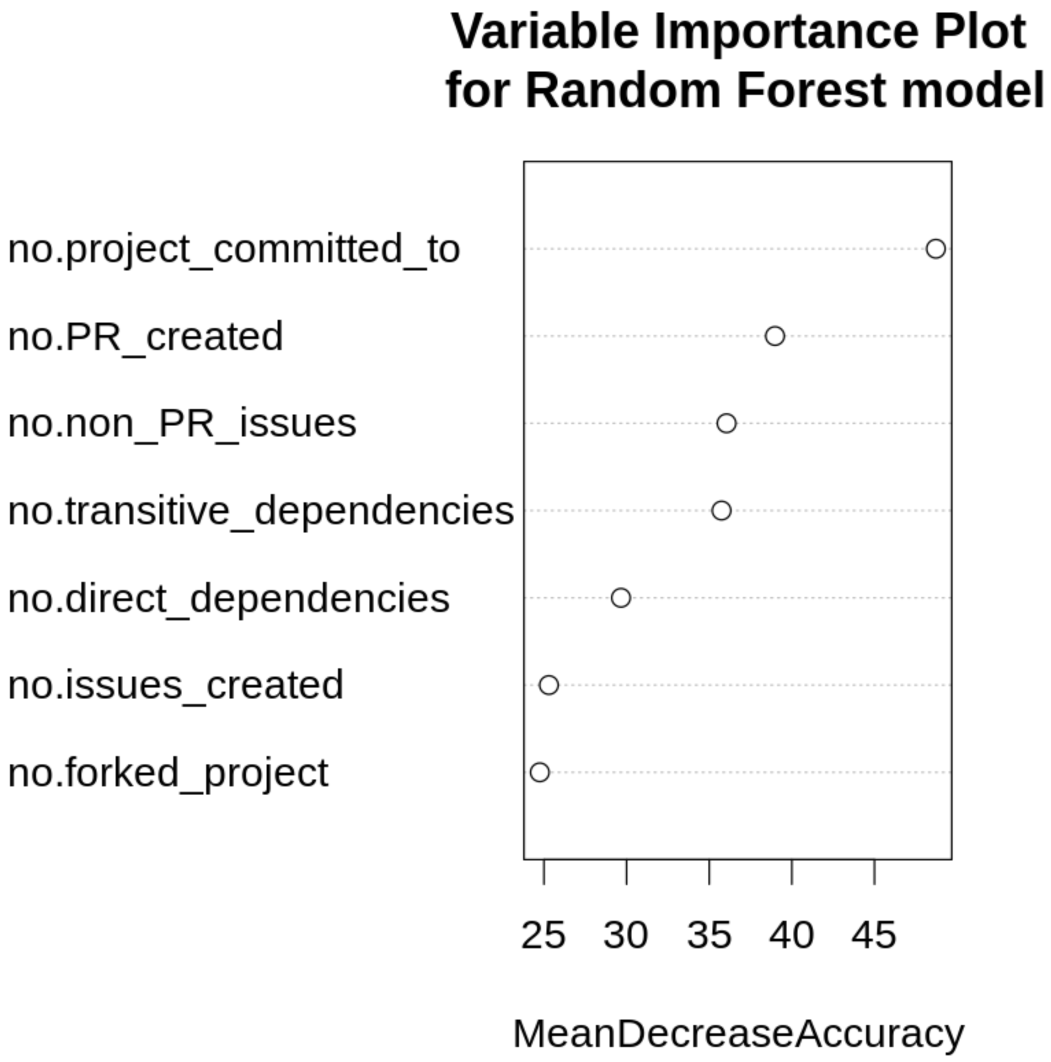


Figure 4-2. Variable Importance plot - Random Forest model

of the two parameters that gives the highest Accuracy, using 10 fold cross-validation. The optimal value of “mtry” was found to be 2, and “ntree” of 500 gave the best performance.

Using the optimal values of the parameters “mtry” and “ntree”, we fitted the Random Forest model on the training data, and tested the performance of the model against the test data. Our model had a sensitivity of 0.62, and it performed relatively worse in terms of specificity (0.47), i.e. it did relatively better in terms of not classifying users without a company affiliation as users with a company affiliation, but a number of users with a company affiliation were wrongly predicted as users without a company affiliation. The value of AUC under the ROC curve was 0.68, and the overall accuracy of our model was 0.70, with a 95% confidence interval between 0.69 and 0.75.

The variable importance plot for our final model is shown in Figure 4-2. The 7 predictors we selected for our final model were (in the same order of importance they appear in Figure 4-2): total no. of Git repositories a user committed to, no. of pull requests created by the user, no. of issues created by the user that are not pull requests, total number of transitive dependencies of all of the user's public repositories, total number of direct dependencies of all of the user's public repositories, total number of issues created by a user, total no. of repositories of the user that are forks of another repository. So, we see that to which layer a user creates an issue or a pull request isn't really important in predicting their company affiliation, but the total activity, the number of projects they committed to, the number of issues, PRs, and non pull request issues they create, and the number of packages the user's public repositories depend on directly and transitively are important in predicting their company affiliation.

To observe how the values of these predictors are different between users with and without a company affiliation, we conducted the one-sided Kolmogorov-Smirnov test to test if the distribution is stochastically larger for one of the groups, for these variables. We found that for users with a company affiliation, the distributions of all of the predictor variables are stochastically larger, i.e. *they create more issues, more PRs, as well as more non PR issues, and they also commit to more projects*, have more dependencies, and more forked projects. Overall, we can say that they have a larger footprint on the NPM ecosystem.

4.7 Discussion

In this section, we discuss the answers we obtained for our research question, and the implications of our findings. **The important findings of our study include:** (1) The distribution patterns of issues and PRs for the NPM ecosystem, which highlight that there are very few cross-project issues and PRs. (2) The presence of distinct user groups, who differ significantly in their participation patterns and amount of activity, and the existence of a large number of users who contribute to packages in level X. (We expected some users like this, since some of their activity may not be public, but we didn't expect so many users would be part of this group.) (3) The shift in participation patterns for the more prolific users, and (4) The possibility of predicting the users' company affiliation by their participation patterns.

Our RQ1 was focused on the distribution of the total number of issues created, and our RQ2 investigated the existence of different groups of users based on the distribution of probabilities of them creating issues at different levels of their respective supply chains. We observed that in terms of creating issues, only 29% belonged to the group who are more likely to create issues for their direct dependencies, but they create around 53% of the total issues. An opposite picture was observed for users who create issues for level X packages, where 68% of the total users are likely to create issues for those packages, but they create around 40% of the issues. This indicates the users who create issues for their direct dependencies are more active. This assumption is further validated when we look at the more prolific users, which shows that more of the prolific users are likely to create issues for their direct dependencies. We observe a similar pattern when we focus on the distribution of PRs and the users who create PRs. However, in this case, we have more users in the group of those more likely to create PRs for level 1 packages. Users creating more issues and PRs for their direct dependencies isn't surprising, since they might face more issues from them and feel more obliged to fix the issues in those packages. However, *the overall trend observed while answering RQ1 and RQ2 led to the following possible implications:* (1) The users who create demand mostly from their direct dependencies are different in nature from those who create demand (issues) from packages outside their supply chain, given they belong to different clusters, and they also differ in their amount of activity. A study looking into the differences between the two groups, their nature, motivation, and reasons for their distinct contribution patterns might give new insights into the NPM ecosystem. (2) We can assume the users who submit PRs are, on an average, more technically proficient than the rest, at least in the given domain. Given the prevalence of low quality issues [117], it might be helpful to predict the quality of an issue or a pull request using the contribution pattern of the user who submitted it.

We observed very few cross-project (level 2+) issues, and even fewer cross-project pull requests. We hypothesize that the reason behind this is a mixture of two factors, (1) the users may not be aware which package is causing some issue they are facing or they do not know how to go about fixing the issue, and (2) they might feel it is not their responsibility to report or fix those issues. A similar situation was reported in [119], which studied the PyPi ecosystem, where a developer said that their experience in trying to fix a bug just two levels upstream was "Extremely Painful", due to their unfamiliarity with the issue reporting system and resolving process, and not being able to convey their problem clearly to the developers in charge. We suspect a similar situation could be true for the NPM ecosystem as well. So, if the reason behind the users not reporting and fixing cross-project issues is more due to the lack of transparency, then this calls for the need of tools and practices that would increase the visibility for the developers beyond the direct dependencies of their code and that would help determine how the packages far in the supply chain might be affecting some issues that they discover when running their code. However, we did observe a small group of users who are more likely to create cross-project issues, both for all the users and the more prolific users, but such a group was not observed when investigating pull requests. Investigating those users might be helpful in formulating a way to increase visibility and streamline the cross-project issue reporting process.

We observed that users with a company affiliation, overall, are more active than the rest, i.e. they contribute to as well as demand more effort from the projects, which might mean that the involvement of different companies is a major driving force behind the growth of the NPM ecosystem. So, if an NPM package gets supported/used by a company, it might be beneficial for the growth of that package, and of the NPM ecosystem overall. Does it indicate the FLOSS community is shifting from its initial structure of software by and for the users [116]? That is a much larger question that needs further study to answer, but our result indicates that companies might have a larger impact on the NPM ecosystem. Using a model similar to ours for identifying the commercial affiliation of the users, and identifying the differences in their contribution patterns might be useful for answering that bigger question.

4.8 Limitations

There are a few limitations to our study that we would like to highlight here. First of all, we only considered the Git repositories with a `package.json` file as JavaScript projects, which is not always true. Also, we extracted the dependency information by looking at the `package.json` and `lerna.json` files, however, looking directly into the source code might have given a much more accurate picture of dependencies. As for dependencies, the dependency map we constructed is for runtime dependency only, i.e. we did not consider the *devDependencies* or any other type of dependencies .

We have assumed in this study that issues create a demand of effort to fix it, and pull-requests can be regarded as contribution of effort by the developers who use a package. While this might be true in general, there is definitely the possibility that the maintainers of a project end up spending a lot of effort fixing some pull-request of poor quality, and, on the other hand, creating a good quality issue report also takes effort from the part of an issue reporter, and the maintainers might have to spend little effort fixing an issue of good quality. However, we believe that our assumption holds true for majority of the cases.

We only looked at the public repositories of the users, for obvious reasons. So, it could be possible that, based on the activity of a user in their private repositories or other projects not shared publicly in Git, some of the packages that we classified as level 2+ for a user could actually be level 1 for them, or some package in level X could actually belong to level 0, 1, or 2+ for that user.

We looked at only 4433 NPM packages, which is less than 0.5% of the total packages in NPM ecosystem, however, given that a huge number of packages are almost never used, we believe this small subset of packages experience bulk of the activity in the ecosystem.

As mentioned before, we extracted the company affiliation information for the users from the information they provided on GitHub. We did not attempt to validate this information from any other source, which leaves the room for some error in classification. However, we believe that more professional developers are likely to provide accurate information about themselves. Another related situation could be that some users actually affiliated to a company never bothered to fill out that information about themselves, leading to a misclassification.

While studying the issues, we did not differentiate between the type of issue, if it is open or closed, and for the pull-requests, if it was merged or not, nor have we checked if the company a user is associated with is one that is centered around OSS development, or a more traditional company.

Our study selected the users based on the criteria that must have created at least one issue, which makes all of our findings are conditional on that selection criteria, and the results may not apply for the entire population of users.

The result we obtained in this paper might not generalize to all types of software ecosystems, since NPM is heavily used by different companies around the world, while many other types of software are not as heavily used.

4.9 Summary: Addressing RT3

The goal of this project was to address the third research topic mentioned in subsection 1.2.1: *"Exploring if the issue contributors of OSS projects depend on the* *project through dependency supply chain, and if the contributors who aren't dependent on the package through the supply chain have a different characteristics than the contributors who are."*

The results of this highlighted that a lot of issues and PRs to various projects are in fact created by developers who do directly depend on the packages, which shows the importance of the supply chain perspective into the ecosystem. However, we also found that some of the issues and PR creators have no visible connection to the packages to which they create issues/ PRs. This warrants further investigation as to how they were exposed to those packages. Another interesting finding is that very few issues/ PRs are created to the packages on which the issue/ PR creators depend on transitively, which is an evidence of the fact that there is a problem related to the lack of visibility, at least for the NPM ecosystem.

Future studies are needed to determine how to increase the visibility, learn from distinct participation patterns, and how these findings apply in other ecosystems.

CHAPTER 5
DO THE CHARACTERISTICS OF A PULL REQUEST CONTRIBUTOR AFFECT THE PROBABILITY OF PULL REQUEST ACCEPTANCE?

5.1 Overview

The dependency between a pull request (PR) contributor's characteristics and the probability of that pull request being accepted is explored in this chapter. Thereby, it tries to answer the fourth research topic posed in subsection 1.2.1 of Chapter 1: "Investigating the effects of the characteristics of individual patch contributors, specifically, their experience and social proximity to the repositories to which they submit PRs, on the chances of their pull requests getting accepted." Effects of a number of other relevant factors are also incorporated in the models used to investigate the relationship in order to control for those variables. Most of the results of this study was from a paper accepted in the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2020 [33].

5.2 Introduction

With the advent of social-coding platforms like GitHub [143], the Pull Request (PR) based development model has become the norm. This model allows developers outside of a project to contribute without compromising the quality of the original project by only merging approved changes to the repository, and was found to be associated with shorter review times and larger numbers of contributors compared to mailing list code contribution models [144].

However, the PR based development model, while clarifying and simplifying the process of contributions from unfamiliar outsiders, does not, by itself, ensure the quality of contributions. This entails the challenges of evaluating the quality of the contributed code (a task that is typically performed by a PR integrator) and maintaining the quality of the project [145]. The PR integrators of popular projects are often overloaded with reviewing multiple pull requests [145], which adds additional complexity to the process.

It has been extensively documented (see, e.g. [117]) that large numbers of low-quality issues may overwhelm the projects and the same is true for pull requests. Being able to gauge the quality of a submitted PR may, therefore, benefit the integrators to prioritize their review process and may, consequently, increase the efficiency of the process. However, PR "quality" has no universal definition and may mean different things in different contexts. We, therefore, chose the ultimate pragmatic indicator: whether it is merged (accepted) or not. It should be based on the contextual knowledge of the integrator at the time of acceptance and should take into account a variety of factors the integrator has to consider when accepting a PR. By doing this we follow a comprehensive treatment of code contribution theory in Rigby et. al. [146] that considers the acceptance rate as one of the most fundamental properties of the peer-review systems. In this study we, therefore, define the quality of a PR by its probability of getting merged.

Several studies (e.g. [147, 148, 149, 150, 135, 151],) investigated the effects of various technical and social factors on the PR acceptance probability. However, those studies primarily contained relatively few ($<$ 100) projects, potentially limiting their generalizability to the entire ecosystem. Different studies highlighted different factors that significantly influence the pull request acceptance probability with no clear answer as to what factors apply broadly. Most studies did not present the relative importance of these factors, nor did they report the functional relationship between the PR acceptance probability and values of the predictors. Finally, prior studies that focused on a set of specific projects did not take into account the ecosystem-wide nature of developer participation (and the corresponding experience).

To address these gaps, we analyzed how different technical and social factors related to the characteristics of a pull request, the PR creator, the repository to which the PR is submitted, the social proximity between the PR creator and the repository, and the PR review phase influence its probability of being accepted by analyzing 470,925 pull requests for 3349 packages (2740 different GitHub repositories), each with more than 10,000 monthly downloads and at least 5 pull requests created against their GitHub repository, from the NPM ecosystem, which is one of the largest open source software ecosystems at present.

Our goal in this study is to deepen the understanding of how various social and technical factors influence the PR acceptance probability at an ecosystem level (where we characterize projects, developers, and pull requests based on measures obtained not just for the specific set of projects, but on the entire ecosystem of projects and developers in the ecosystem-wide software supply chain). Knowing how these measures, including the contextual factors that may be unique to individual projects or sub-ecosystems, influence PR acceptance may help the PR creators, integrators, and also the tool designers who design pull request evaluation interfaces. Specifically, the tool designers might choose to make important signals more readily available to the PR creators, who can better format their individual pull requests to have a better chance of having their contributions accepted, and to the PR integrators, who can look for those signals to gauge the quality of the pull requests they are evaluating. [1]

Our key contributions in this study include the conceptual replication of prior findings for the ecosystem-level model and data that relates the probability of acceptance to the technical characteristics of a PR, the track record of a PR creator in terms of having their pull requests accepted and their social proximity to the repository to which the PR is submitted, and measures describing the characteristics of the PR review phase. We also introduce new ecosystem-level measures: *the overall experience of the PR creator across all OSS projects, the leniency of a repository in terms of accepting pull requests,* and *the presence of a dependency between the repository the PR creator submitted the pull request to and the projects they previously contributed to* and show that they have a significant impact. A predictive model with these measures achieved an AUC-ROC value of 0.94. Finally, we observed nonlinear dependencies between the PR acceptance probability and most of these measures (see Section 5.6.4), suggesting a variety of potential mechanisms and other factors like the presence of bots in the dataset that appear to drive PR acceptance in different contexts. We have also created a dataset with the curated data of the pull request properties we measured, along with their descriptions, a code snippet for creating a Random Forest model using the data, and we also included the final Random Forest model we used for predicting pull request acceptance. The dataset is available at: DOI 10.5281/zenodo.3858046 [152].

5.3 Related Works

There have been a good number of studies on pull requests, which investigated different questions related to the dynamics of PR creation and acceptance, e.g. estimation of PR completion times [153], finding the right evaluator for a particular PR [154, 155, 156, 157], predicting whether a PR will see any activity within a given time window [158] etc.

[1]We are not performing a causal analysis, thus, we do not intend to make the integrators believe that a PR is bad just because it doesn't match the characteristics of a typically acceptable PR, or vice versa.

There are other studies that explore the perspective of the PR creators [159] and the PR integrators [145], and list the challenges and practices in PR creation and merging scenarios.

A number of studies describe various factors that influence the chance of a PR getting accepted, like [148], which advocates using association rules to find the important factors, and found that the acceptance rates vary with the language the repository is written in, and also that having fewer commits, no additions, some deletions, some changed files, and the author having created a PR before and/or being part of the core team increase the chance of getting a PR accepted; [147], which indicates smaller pull requests are more likely to get accepted; [160], which shows previously established track records of the contributors, availability and workload of the evaluators, and continuous integration based automated testing etc. have an impact on the latency of PR evaluation; [149], which examined the effects of developer experience, language, calendar time etc. on the PR acceptance; [161], which analyzed the association of various technical and social measures, e.g. adherence to contribution norms, social proximity between the creator and the project, amount of discussion around the PR, number of followers of the PR creator, and popularity of the repository to which the PR is submitted, with the likelihood of PR acceptance. There are a number of case studies that discuss the PR acceptance scenario in various OSS projects, like the Linux kernel [150], Firefox [151, 135], Apache [135] etc. All of the studies mentioned above, except [161], focused on a few (<100) projects, so the general applicability of their findings wasn't verified at an ecosystem level. As for [161], while it studied a large number of (12,482) GitHub projects and reported the significance of various social and technical factors by the odds ratio measure (which itself has a few drawbacks, e.g. odds ratio can overstate the effect size [162] and can lead to misleading implications, especially when events, e.g. PR acceptance in this situation, are very common or very rare [162, 163]), it doesn't show the relative importance of the factors involved, nor does it show how the PR acceptance probability varies with the values of these factors.

In our study, we aim to add to the current body of knowledge about this topic by addressing the limitations of the previous studies, specifically by showing the relative importance of different factors, identifying how different values of each of these factors affect the PR acceptance probability, and verify the applicability of our findings for the NPM ecosystem.

5.4 Research Questions

Our first research question focuses on identifying various social and technical factors that might affect the PR acceptance probability, and testing the significance of those factors for our dataset:

RQ1: What are the various Social and Technical factors that affect PR acceptance probability?

We formulated several hypotheses for addressing this research question. In particular, our review of the related literature revealed a number of factors that might potentially affect PR acceptance probability, viz.:

H1 *The technical characteristics of a PR*, described by its size and factors like inclusion of test code and issue fixes, *have a significant impact on PR acceptance probability,* with existing literature (e.g. [147, 161]) suggesting that smaller PRs are more likely to be accepted as they are easier to review and more likely to involve a single task.

H2 *Social proximity between the PR creator and the repository to which it is created also increases the PR acceptance probability* (see, e.g. [161, 160]).

H3 *A previous track record of the PR creator in getting their contributions accepted (in the same ecosystem) increases the PR acceptance probability* (see, e.g. [158]).

H4 *Characteristics of the PR review phase can have an impact on the probability of PR acceptance*, e.g. a higher amount of discussion around the PR was found to have a negative effect on PR acceptance [161].

In addition to considering the particular measures related to these factors described in earlier studies, we hypothesized that a few other measures might also affect the PR acceptance probability:

H5 *A more experienced PR creator will have a higher chance of getting their PRs accepted*, because experienced developers would create better PRs, e.g, via H1. The experience may also be associated with their reputation, thus potentially reducing the social distance via H2. It is worth mentioning that [149] also described a variable named developer experience that can affect the PR acceptance probability, however, while they simply referred to the duration of their job experience, we refer to a set of more specific measures, viz. the number of commits made by them and the number of projects they have contributed to in the entire OSS ecosystem.

H6 *PRs to a repository, which has a track record of being more lenient in terms of accepting PRs, are more likely to be accepted*, as evidenced by the fact that getting a PR accepted in projects like Linux kernel tends to be harder due to the project practices of placing stringent requirements on the PRs.

H7 *If any of the projects the PR creator previously contributed to depend on the repository to which the PR is being created, it is more likely to be accepted*, since it may be in the self-interest of the PR creator to make the quality of the patch better if their projects depend on the repository. Alternatively, depending on a specific repository may increase their expertise on that project.

Our first research question is addressed by testing the validity of the above-mentioned hypotheses for our dataset. Specifically, testing H1-H4 represents a conceptual replication of earlier work on a different dataset with ecosystem-wide operationalizations of the original measures.

We also want to investigate the relative importance of the factors mentioned above in predicting if a PR will be accepted, since such knowledge would help the developers to know and prioritize the aspects they should focus on to get their contributions accepted (for the PR creators), or to gauge the quality of a submitted PR (for the integrators), or trying to decide which signals to make available to the parties involved (for tool designers).
RQ2: What are the relative importance of different measures related to the factors found to be significant while trying to predict whether a PR will be accepted?

Finally, we want to find out the functional relationship between the PR acceptance probability and the aforementioned predictors. Should developers try to maximize or minimize them, how important such modification might be in increasing the chances, or is there a "sweet spot" to target? This can also help the researchers in gaining a better understanding of the complex dynamics of the process of pull request creation and acceptance.
RQ3: How does the PR acceptance probability vary with different values of the predictors?

5.5 Methodology

In this section we describe the data used in this study, the process of data collection and preprocessing, and the measures used for describing the factors mentioned above.

5.5.1 Pull Request Selection

To conduct an empirical study investigating how different technical and social factors affect the PR quality we chose to focus on the node package manager (NPM) ecosystem because of its size, popularity among software developers at present, and the availability of data. NPM is a package manager of JavaScript packages, and is one of the largest OSS communities at present, with over a million different packages and millions of users (estimated 4 million in 2016 [127], and about 4000 new users on an average day[2]). NPM is used heavily by companies as well. According to the NPM website[3], all 500 of the Fortune 500 companies use NPM, and they claim that: " *Every company with a website uses npm, from small development shops to the largest enterprises in the world.*" However, most packages in NPM are not widely used and have limited or no pull requests. We, therefore, focused on the NPM packages with over 10,000 monthly downloads (the "popular" packages) since January, 2018 (to ensure that they maintained their popularity for a sustained period of time), that also has an active GitHub repository with at least 5 pull requests created against it. We chose to remove packages with very few pull requests because the effects of the repository related measures on PR acceptance might give misleading results for them. In addition, we chose to only focus on the pull requests that were created on or before April, 2019 and were marked as "closed", to ensure that they have had sufficient time to be resolved. This way, we were left with 470,925 pull requests that were created against 2740 GitHub repositories of 3349 NPM packages. These pull requests were created by 79,128 unique GitHub users, and were evaluated by 3633 unique integrators.

5.5.2 Selection of Measures for Verifying the Hypotheses H1-H7

We constructed the measures that could describe the factors we suspect might affect the probability of a PR being accepted from our collected dataset by consulting prior work, monitoring the discussion on different online communities, and from our experience. A detailed description of the variables is available in Table 5-1.

5.5.2.1 Outcome Measure

The outcome measure, i.e. the variable that we are trying to predict is whether a pull request is accepted or not, and is essentially a dichotomous variable.

[2]https://twitter.com/seldo/status/880271676675547136
[3]https://www.npmjs.com/

Table 5-1. Detailed Definition of the selected Measures and their Descriptive Statistics: Median, Mean, 5% and 95% values for continuous variables, number of Yes/No values (represented by 0 & 1 in the data, respectively) for binary variables (highlighted in yellow). Measures marked with asterisk(*) were not used in any previous study.

Underlying Factors (Hypothesis)	**Measure**	**Variable Description**	**5%**	**Median**	**Mean**	**95%**
Pull Request Characteristics (H1)	additions	Number of lines added in the Pull Request	1	12	703	619
	deletions	Number of lines deleted in the Pull Request	0	2	385	248
	commits	Number of commits in the Pull Request	1	1	4	7
	changed_files	Number of files modified in the Pull Request	1	2	10	17
	contain_issue_fix	If the Pull Request contained a fix for an issue	**No (0)**: 267811 (57%), **Yes (1)**: 203114 (43%)			
	contain_test_code	If the Pull Request contained test code	**No (0)**: 360522 (77%) , **Yes (1)**: 110403 (23%)			
Social connection between PR creator and the repository (H2)	user_accepted_repo	If the PR creator had a contribution accepted in the repository earlier	**No (0)**: 198087 (42%), **Yes (1)**: 272838 (58%)			
Track record of the PR creator (H3)	creator_submitted*	Number of PRs submitted by the PR creator across NPM projects	0	12	282	1043
	creator_accepted	Fraction of PRs submitted by the PR creator accepted across NPM projects	0	0.64	0.53	1.00
Characteristics of the pull request review phase(H4)	comments	Number of discussion comments against the Pull Request	0	2	3	11
	review_comments	Number of code review comments against the Pull Request	0	0	1	6
	age*	Seconds between PR creation and PR closure	231	100*1e3	651*1e3	2.5*1e6
PR creator experience (H5)	creator_total_commits*	Total number of commits made by the PR creator across git Projects	4	786	9847	12,386
	creator_total_projects*	Total number of projects the PR creator contributed to across git Projects	3	1632	6481	31,880
Repository characteristics (H6)	repo_submitted*	Number of PRs submitted against the repository	9	787	4787	30,270
	repo_accepted*	Fraction of the submitted PRs accepted by the repository	0.1	0.70	0.63	0.91
Dependency between PR creator's projects and the package (H7)	dependency*	If any of the repositories the PR creator contributed to depend on the package	**No (0)**: 82215 (17%), **Yes (1)**: 388710 (83%)			

5.5.2.2 Measures related to the technical characteristics of a PR

In addition to the number of commits, lines added, lines deleted, changed files in the PR, all of which are continuous variables related to its size, we also considered two dichotomous variables, describing if the PR contained any test code and whether the PR description explicitly mentioned that it contains an issue fix, as the measures that describe the technical characteristics of a pull request.

5.5.2.3 Measures related to the social proximity between the PR creator and the repository to which the PR is being created

Following the recommendation by [161], we considered a dichotomous variable, describing if the PR creator had previously created a PR against the repository which was accepted to be indicative of the social proximity between the PR creator and the repository.

It is worth mentioning that the PR creator's association with the repository where the PR was submitted was found to be one of the most important predictors by [161]. However, we decided not to use this variable in our final dataset, since the value of this field can be updated retroactively (e.g. a PR creator, who had no association with a project when first submitting a PR, might become a *member* later, and the corresponding field in the first PR might be updated after its acceptance/ rejection), and we have no way to know the creator's association at the time when the PR was submitted, or to verify that the affiliation wasn't updated retroactively, which would be required to faithfully reconstruct the data as it were at the time the PR was created. In fact, we suspect the affiliation is indeed updated retroactively, since we found that all PR creators with an accepted contribution to a repository have some association with it. We, therefore, suspect that its importance in prior work could be due to the so called data leakage [164], when the information leaked from the "future" makes prediction models misleadingly optimistic.

5.5.2.4 Measures related to the track record of the PR creator

After considering the measures used by previous studies to describe the track record of the PR creator, we decided to focus on two variables that we believe adequately captures their track record: the number of pull requests created by the PR creator (before creating the one under consideration) in the same ecosystem and what fraction of those have been accepted. The first measure wasn't actually used in any previous study, but we believed that it would also be an important measure and decided to include it.

5.5.2.5 Measures describing the PR review phase

We considered the following variables to be descriptive of the PR review phase: the number of comments and review comments, along with the age of the PR, which wasn't used in any previous study, but should also reflect the complexity of the evaluation process for the pull request.

5.5.2.6 Measures describing the experience of the PR creator

As described in Section 5.4, we are looking for the specific measures describing the overall experience of the PR creator, viz. the total number of commits they created and the total number of projects they have contributed to across all projects that use git. We believe that the overall experience of the developers should be reflected in the pull requests they create, for example, a developer with a considerable amount of experience in contributing to different OSS projects would likely create a good quality pull request even if it is the first time they are trying to contribute to it.

5.5.2.7 Measures reflecting the pull requests received and accepted by a repository

As described in Section 5.4, we believe the policy of a repository about accepting pull requests should have an impact of PR acceptance probability, which should be captured by the following two variables: the number of pull requests submitted to the repository before the submission of the one under consideration, which should reflect the popularity of the repository and how much workload the integrators might have, and the fraction of pull requests that are accepted, which should be reflective of the leniency of repository towards accepting contributions.

5.5.2.8 Measure describing if the PR creator depends on the package to which they are submitting a PR

This measure is a dichotomous variable showing if any of the projects the PR creator has contributed to prior to submitting the PR under consideration depend on the repository/ package to which the PR is submitted.

5.5.3 Data Collection

To be able to identify the pull requests that adhere to our criteria, as mentioned above, we first needed the list of all NPM packages that have more than 10,000 downloads per month and a GitHub repository with at least 5 pull requests. This was obtained from the `npms.io` website using their API. [4] The associated GitHub repository URLs were collected from the metadata information of these packages, which was obtained by using a “follower" script, as described in NPM’s GitHub repository. [5] After filtering for our criteria that the NPM package must have more than 10,000 monthly downloads (since January, 2018) and an active GitHub repository, we were left with 4218 different NPM packages.

[4]https://api.npms.io/v2/package/[package-name]

[5]https://github.com/npm/registry/blob/master/docs/follower.md

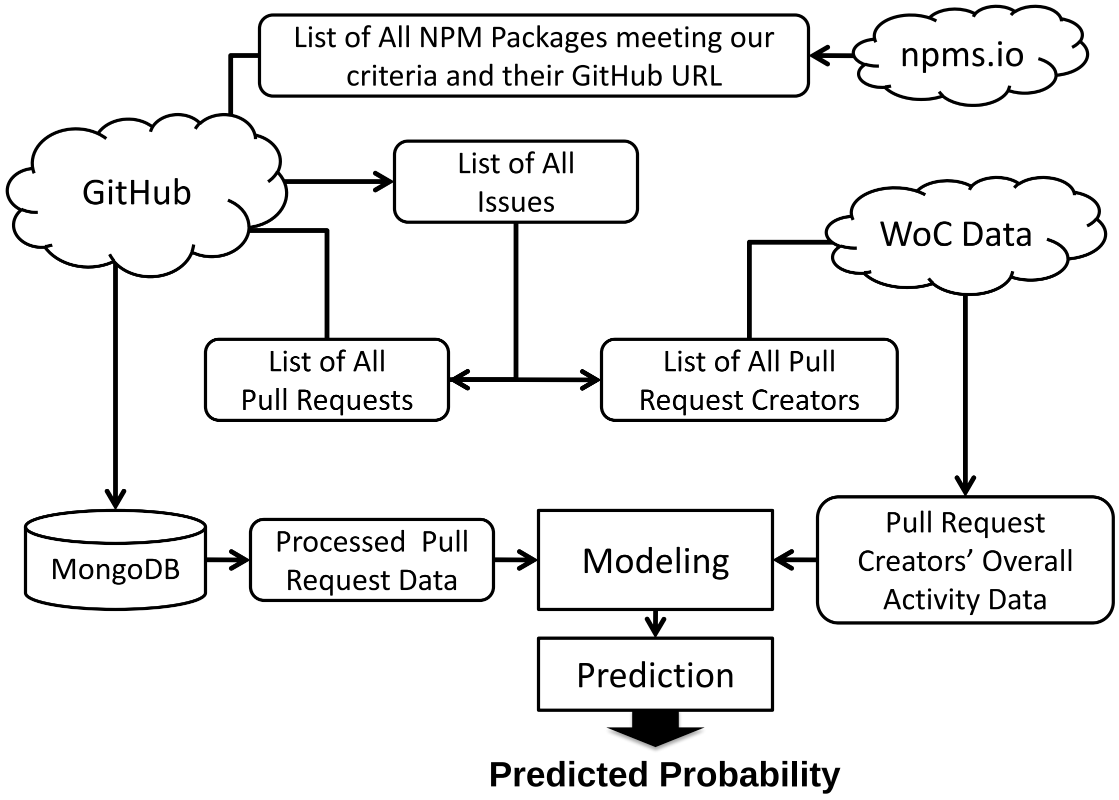

List of All NPM Packages meeting our criteria and their GitHub URL
npms.io
GitHub
List of All Issues
WoC Data
List of All Pull Requests
List of All Pull Request Creators
MongoDB
Processed Pull Request Data
Modeling
Pull Request Creators' Overall Activity Data
Prediction
Predicted Probability

Figure 5-1. The Data Collection and Modeling Architecture.

Next, we needed the list of all pull requests for these NPM packages. To obtain this, we first collected all the issues associated with these NPM packages, since GitHub considers pull requests as issues, and then identified the issues that have an associated patch, i.e. the ones that are Pull Requests. The list of all issues for the packages was obtained using the GitHub API for issues[6], using the `state=all` flag. We identified 483,988 pull requests for all the issues for these 4218 packages. It is worth mentioning here that sometimes more than one NPM package can have the same associated GitHub repository, e.g. all TypeScript NPM packages (starting with "@types/", like @types/jasmine, @types/q, @types/selenium-webdriver etc.) refer to GitHub repository "DefinitelyTyped/DefinitelyTyped". To avoid double-counting and further confusion, we saved the issues keying on the repository instead of the package name, though we also saved the list of packages associated with a repository. We found that there are 3601 unique repositories associated with these 4218 packages. We

[6]https://developer.github.com/v3/issues/

further filtered this dataset to only have the repositories that had at least 5 pull requests, and found that 2740 repositories, associated with 3349 NPM packages, match this criteria. Then, we obtained the data on all the pull requests for these repositories from GitHub using the API.[7] This data was stored in a local MongoDB database. We used a Python script to extract the data from this database and process it into a CSV file that was used for modeling. We further filtered out all pull requests that were not marked "closed", since we are only interested in looking at the already resolved pull requests.

The data on the PR creators' overall activity across all projects that use git were obtained from the World of Code (WoC) data [30]. WoC is a prototype of an updatable and expandable infrastructure to support research and tools that rely on version control data from the entirety of open source projects that use git. The data in WoC is stored in the form of different types of maps between different git objects, e.g. there are maps between commit authors and commits, commits and projects etc. The detailed description of this dataset is available in [30] and the project website. [8] Specifically, we used this dataset to compile the profiles of PR authors. Since the author ID used in WoC is different from the GitHub ID of developers (WoC author IDs comprise the name and email address of the developers), we identified the PR authors by first obtaining the commits they included in their pull requests, and then by identifying the author IDs for these commits in WoC using commit to author maps. Then we identified all the remaining commits for these authors using the author to commit map. That full set of commits for each author was used to count the number of projects they contributed to. To construct the relevant measures for the PR acceptance prediction, we only used the commits made by the PR author prior to the creation of the PR, thus reconstructing the state of affairs as it existed at the time of PR creation.

5.5.4 Variable Construction

As shown in Table 5-1, we listed 17 different measures related to the hypotheses we proposed in Section 5.4. The data for 6 of those measures, *additions, deletions, commits, changed_files, comments,* and *review_comments*, were directly obtained from the data collected from GitHub API. The data on two measures, *creator_total_commits* and *creator_total_projects*, were obtained directly from the WoC dataset. To calculate the *dependency* measure, we considered all the commits made by the PR creator and, using

[7]https://developer.github.com/v3/pulls/
[8]https://bitbucket.org/swsc/overview/src/master/

those, find out which projects they have contributed to. Then we check the dependencies of all those projects to see if any of them list the package as a dependency.

Calculation for the *age* variable was relatively straightforward, we simply counted the seconds between the time of PR creation and the time of PR closure. For determining whether the PR contained any test code or mentions fixing an issue, we looked at its description and populated the measures *contain_test_code* and *contain_issue_fix* based on whether the description of the submitted PR mentioned including test code (we looked for the phrase "test code" and a few of its variations in the description) and fixing an issue (we checked if the description has one of the words signifying "fix", e.g. "fix", "resolve" etc. and an issue, e.g. the word "issue", the symbol "#" followed by a number etc.).

For calculating the remaining variables, we sorted our whole dataset by PR creation times and counted the cumulative number of pull requests created by each PR creator and against each repository and kept track of the fact of whether they were accepted or not. Using this cumulative data, we calculated the values for the variables *creator_submitted, creator_accepted, repo_submitted, repo_accepted,* and *user_accepted_repo*.

Since the values of all continuous variables except *creator_accepted* and *repo_accepted* were found to be heavily skewed, we converted them into log scale for modeling purposes.

5.5.5 Analysis Method

We used logistic regression for verifying which variables have a significant impact on PR acceptance probability, and used Random Forest method for measuring the predictive performance of our model and ranking the measures by importance. The variations of PR acceptance probability with the values of these measures were identified using partial dependence plots [165].

5.6 Results

5.6.1 General Background

Our study focused on 3349 NPM packages (2740 unique GitHub repositories) with more than 10,000 monthly downloads since January, 2018, an active GitHub repository, and at least 5 pull requests created against them. We collected 470,925 pull-requests, which were created by 79,128 unique GitHub users, and were evaluated by 3633 unique integrators.

Of these pull requests, 290,058 (61.6%) were accepted (merged into the codebase), which were created by 47,099 unique GitHub users (59.5% of all PR creators). 87 repositories (3% of the ones under consideration), which had a total of 981 pull requests created against them, didn't accept any of the pull requests. Conversely, 124 (4.5% of total) repositories, who received 4230 pull requests in total, accepted all of them.

Table 5-2. Regression Model showing the significance of the measures listed in Table 5-1 in explaining PR acceptance. P-values less than 0.0001 are shown as 0. Measures not found to be significant are highlighted in Red. Dichotomous variables are shown in blue.

Underlying Factors (Hypothesis)	Measure	Coefficient ± Std. Error	p-Value
	(Intercept)	0.3830 ± 0.0255	0
Pull Request Characteristics (H1)	additions	-0.0168 ± 0.0030	0
	deletions	-0.0010 ± 0.0029	0.7332
	commits	-0.2475 ± 0.0076	0
	changed_files	0.0219 ± 0.0073	0.0026
	contain_issue_fix:1	0.0338 ± 0.0077	0
	contain_test_code:1	0.1046 ± 0.1236	0.3976
Social connection between PR creator and the repository (H2)	user_accepted_repo:1	0.7921 ± 0.0111	0
Track record of the PR creator (H3)	creator_submitted	-0.1371 ± 0.0028	0
	creator_accepted	1.3590 ± 0.0128	0
Characteristics of the pull request review phase (H4)	comments	-0.4519 ± 0.0054	0
	review_comments	0.2785 ± 0.0059	0
	age	-0.2100 ± 0.0015	0
PR creator experience (H5)	creator_total_commits	0.0115 ± 0.0027	0
	creator_total_projects	0.0256 ± 0.0023	0
Repository characteristics (H6)	repo_submitted	0.0071 ± 0.0017	0
	repo_accepted	3.3468 ± 0.0174	0
Dependency between PR creator's projects and the package (H7)	dependency:1	0.0317 ± 0.0099	0.0014

5.6.2 Testing the significance of the measures

In order to find out which variables have a significant effect on PR acceptance, we used a logistic regression model, the result of which is presented in Table 5-2. We checked the variance inflation factors (VIF) for this model to ensure there is no multicollinearity problem and found the maximum value of VIF to be 3.1, which is within acceptable threshold. Two out of the total 17 measures used in the model as predictors were found to be insignificant, both of which are related to the technical characteristics of a PR.

5.6.2.1 Examining H1

The measure showing whether the PR included test code was found to be significant in [161], but turned out to be insignificant for our dataset. The size of a pull request was also reported to be significant in [161, 147], who looked at the number of lines changed. We described the lines changed by two different variables: number of lines added and number of lines deleted. While the number of lines added was found to be a significant predictor, the number of lines deleted was found to be insignificant for our dataset. Two other variables we hypothesized to be descriptive of the pull request's technical characteristics: the number of commits in the PR and whether the PR description explicitly mentioned fixing an issue were both found to be significant predictors. Explicit mention of an issue fix in the PR description was found to increase the chance of a PR being accepted. The number of lines added and the number of commits negatively affect the PR acceptance probability, which suggests that, in general, smaller patches have a higher chance of being accepted, and supports the findings of [161, 147]. However, we can see from Table 5-2 that the number of changed files seems to have a positive effect on PR acceptance probability, which is contrary to what was reported in [161]. Upon further inspection, the actual scenario turned out to be a bit more complex, which is discussed in detail in Section 5.6.4. Although a couple of measures related to this factor were insignificant, the technical characteristics of a PR as a whole were seen to have a significant impact, though only some of the relationships reported previously could be replicated in our case.

5.6.2.2 Examining H2

We found the measure describing the social proximity between the PR creator and the repository to be significant. Having a contribution accepted in the project earlier had a positive influence on PR acceptance probability, similar to what was reported by [161]. This finding increase the generalizability of the social proximity as an important predictor of PR acceptance.

5.6.2.3 Examining H3

The number of pull requests submitted by the PR creator (before the one under consideration) and the fraction of those pull requests that were accepted both proved to be significant predictors for explaining PR acceptance probability. While the total number of pull requests submitted earlier had a negative influence, possibly due to the presence of relatively inexperienced PR creators in the dataset whose submissions aren't accepted, a higher fraction of accepted pull requests had a strong positive influence on the probability of PR acceptance, showing the importance of a good track record, as was also reported in [158].

5.6.2.4 Examining H4

The variables describing the characteristics of the PR review phase: the number of discussion comments, the number of code review comments, and the age of the PR were all found to be significant predictors. The number of discussion comments was observed to have a negative influence on PR acceptance probability, similar to what was observed by [161]. Although the actual situation might be a bit complex, as described in Section 5.6.4, the number of code review comments was found to have a positive influence on PR acceptance overall. Therefore, the general statement made by [161] that highly discussed contributions are less likely to be accepted seems to come with a caveat: it is true if we are referring to the discussion comments, but false if we talk about the code review comments. The age of a PR seems to have a negative influence on the acceptance probability in general, indicating that good quality pull requests are accepted pretty quickly in the NPM ecosystem.

5.6.2.5 Examining H5

The overall experience of a developer was seen to have a significant effect on the probability of their pull request contributions being accepted, with the probability of acceptance increasing with the number of commits made by the PR creators and the number of projects they have contributed to. While this appear intuitive, we hope that other studies on different ecosystems would confirm these findings.

5.6.2.6 Examining H6

The characteristics of the repository were also found to have a significant impact on PR acceptance probability. The number of pull requests submitted to a repository as well as the fraction of pull requests accepted by it had a positive impact on PR acceptance probability. The faction of pull requests accepted was the most influential variable we had (it had the highest z-score). This observation leads us to believe that our hypothesis, that the leniency of a repository towards accepting pull requests has a significant impact on PR acceptance, holds. It further strengthens the credibility of the assumption that contextual factors, especially ones that are repository-specific, might play a substantial role.

5.6.2.7 Examining H7

The hypothesis that if any of the projects the PR creator contributed to has a dependency on the package to which the creator submitted a PR, it has a higher chance of being accepted also holds according to our model, as can be observed from Table 5-2.

Overall, we found that all of the null hypotheses, denoted by H1.0-H7.0, (null hypothesis is that the postulated factors have no effect on PR acceptance) corresponding to the ones we presented (H1-H7) were rejected, which allows us answer the first Research Question we posed. Although we can see the direction of the relationships between the variables and PR acceptance probability may not always be intuitive as shown in Table 5-2, that aspect is examined in more detail later (Section 5.6.4). While, generally, we got results consistent with prior literature, there were a number of exceptions suggesting that further studies may be needed to resolve these inconsistencies.

Answering RQ1: All of the null hypotheses (H1.0-H7.0) we posed were rejected on our dataset, indicating the factors found to be significant in earlier studies (H1-H4) as well as the ones we proposed (H5-H7) have significant influence on PR acceptance.

5.6.3 Relative Importance of the measures in predicting PR acceptance

To gauge their predictive power and determine their relative importance for predicting if a PR will be accepted we created a Random Forest model with the 15 predictors found to be significant from our earlier analysis. Although our observations are independent of each other, there is an underlying element of time in the whole dataset. Therefore, to ensure that there is no data leakage concerns, we decided to divide the data such that the model is trained on 70% of the pull requests that are created before the rest, and we tried to predict the acceptance of the remaining 30% of the pull requests. We repeated the process 1000 times to ensure there is no significant variation in performance and the relative importance of the predictors.

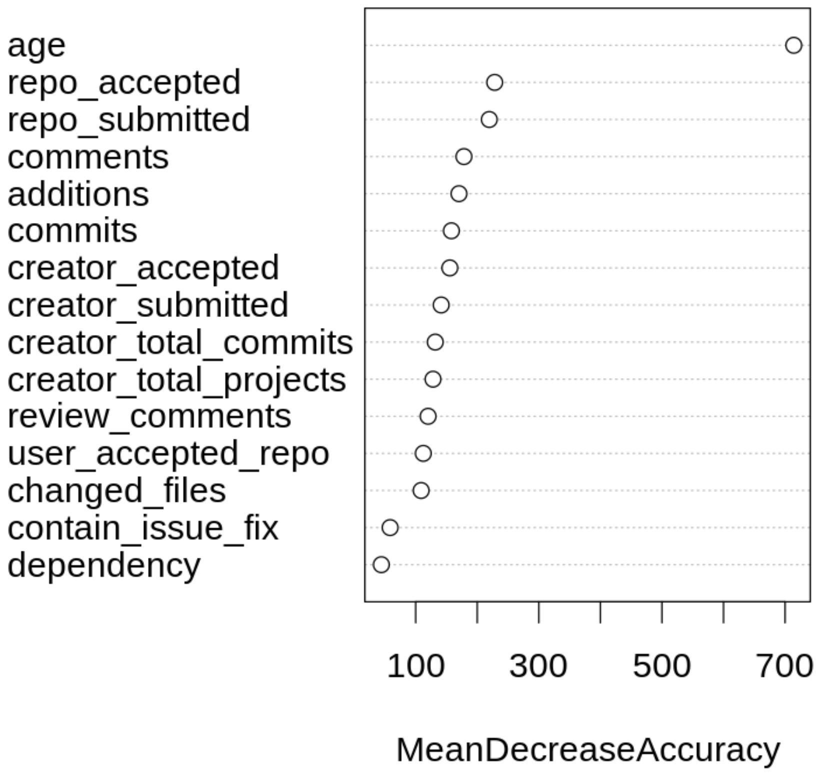


Figure 5-2. Variable Importance Plot from the Random Forest model for predicting Pull Request Acceptance

Our model achieved an AUC-ROC of 0.94, with the values of sensitivity and specificity being 0.69 and 0.76 respectively. The variable importance plot, which shows the most common order of the relative importance of the variables in terms of mean decrease in accuracy, is presented in Figure 5-2. This ordering lets us understand the relative importance of different measures in predicting PR acceptance and answer RQ2.

Answering RQ2: The age of the pull request is the most important variable for predicting PR acceptance, followed by the two repository characteristics measures and the measures related to the size of the PR. Measures describing the review phase (other than age) and the track record and experience of the PR creator also turn out to be relatively important. Comparatively, measures related to the social proximity between the creator and the repository, the measure of PR characteristics not related to its size, and the one describing if there is a dependency between the creator's projects and the package turn out to have lower importance in predicting PR acceptance.

5.6.4 Variation of the PR Acceptance Probability with the Predictors' values

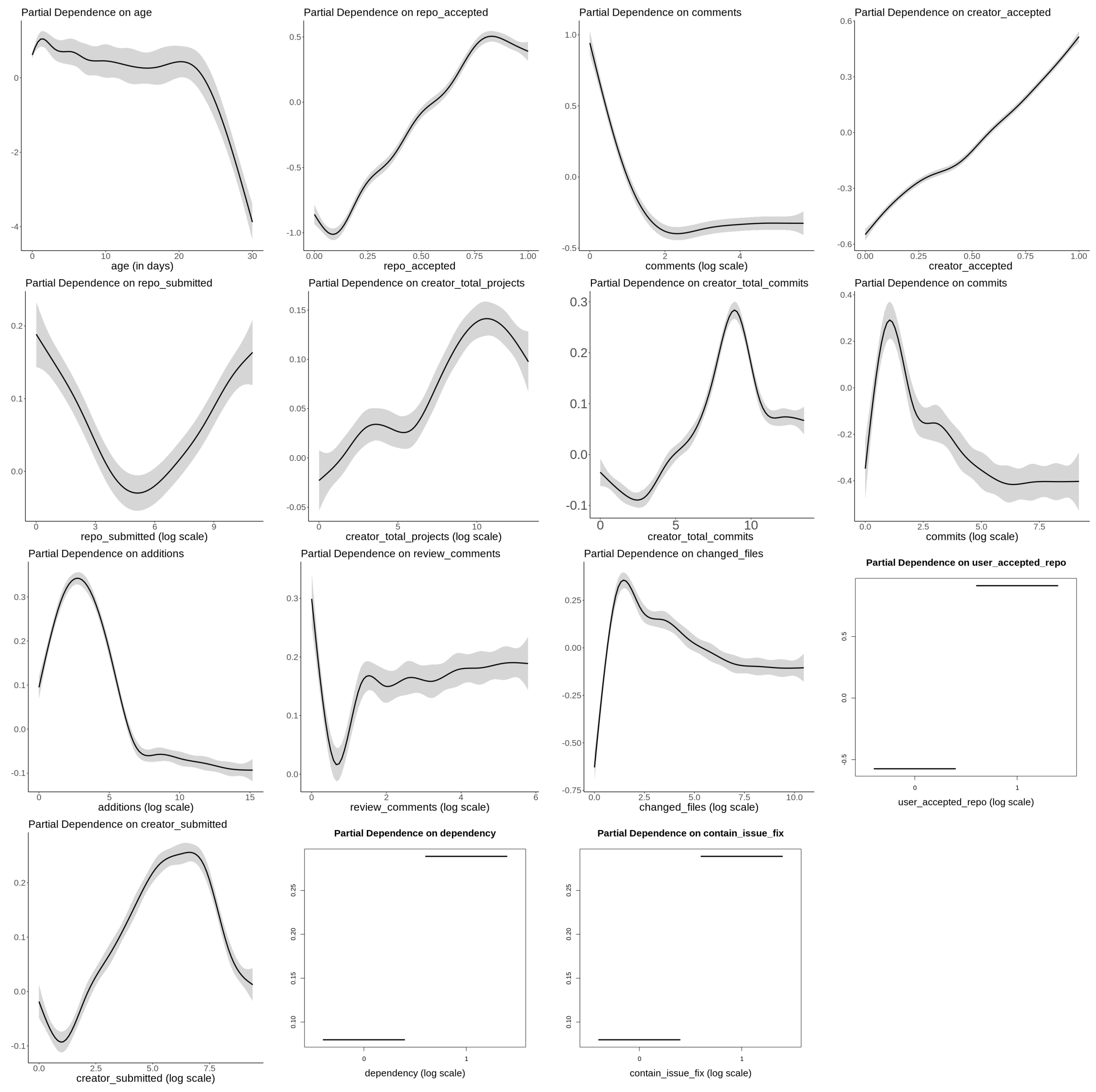
To identify how the probability of PR acceptance varies with the values of the 15 measures found to be significant, we generated partial dependence plots, [165, 168] which are graphical depictions of the marginal effects of these variables on the probability of PR acceptance, from the Random Forest model we created. In the X axes of a plots we have the values of the independent variables and the Y axes of the plots show the relative logit contribution of the variable on the class probability [168] (probability that a PR was merged, in our case) from the perspective of the model, i.e. negative values (in the Y-axis) mean that the positive class is less likely (i.e. it is less likely that a PR would be accepted, in our case) for that value of the independent variable (X-axis) according to the model and vice versa. These plots can shed light into the dynamics of PR creation and acceptance, and would be helpful for both the PR creators and the integrators for understanding how to improve the quality of PRs being submitted and accepted.

The resultant plots for the different measures are presented in a tabular format in Figure 5-3. We generated the plots with the "randomForest" package in R. However, we observed that the plots are not entirely smooth for a few of the measures, so, in order to be able to interpret the results better, we decided to smooth the plots using generalized additive models, which was achieved by using the `geom_smooth` function of the "ggplot2" package in R with the option `method = "gam"`. The plots in Figure 5-3 are arranged by decreasing importance, as observed from Figure 5-2.

5.6.4.1 Variation with pull request age

As observed from Figure 5-3, the probability of a pull request being accepted drops steadily with time. We also observe a catastrophic drop in PR acceptance probability as

Figure 5-3. Partial Dependence plots for the 15 measures from the Random Forest model for predicting PR acceptance probability. The plots were generated using "randomForest" [166] package in R, and smoothed for ease of interpretation with "ggplot2" [167] package in R using generalized additive models (GAM).



it gets older than 20 days. This suggests that the pull request integrators in the NPM ecosystem tend to be quite responsive and efficient in handling pull requests and older PRs may even not be considered. It may also be reflective of the rapid development in NPM ecosystem which makes it harder for the older PRs to get merged with the main development branch.

5.6.4.2 Variation with fraction of pull requests accepted by a repository

We see that the repositories that accept a larger fraction of the pull requests submitted against them are more likely to accept pull requests in the future, which could indicate a more lenient policy of PR acceptance and/or the integrators' willingness to accept contributions from other developers. However, we see that repositories that have very high or very low acceptance rate seem to deviate from this trend, but we suspect that this is due to the "cold start" problem: as a repository just starts getting pull requests, the fraction of those they accept or reject changes the value of this measure dramatically.

5.6.4.3 Variation with the number of discussion comments

We observe that the probability of PR acceptance steadily drops as the number of discussion comments increases. However, as the number of comments go beyond 7 (2.08 in the partial dependence plot in Figure 5-3), the drop in probability gets saturated. This indicates that if the value added by a PR is "obvious", it is more likely to be accepted.

5.6.4.4 Variation with the fraction of pull requests created by a PR creator that are accepted

We observe that the probability of a PR being accepted increases steadily with the value of this measure, which highlights that a good track record of a PR creator has a strong positive influence on the probability of their pull requests being accepted.

5.6.4.5 Variation with total number of pull requests submitted against a repository

We observe from the partial dependence plot of this measure that repositories that either have a small or a large number of PRs created against them are more likely to accept one compared to the repositories that get a moderate number of pull requests. This may be because for the repositories that get a smaller number of pull requests, it is easier for them to evaluate those requests and work with the PR creators to get the contributions accepted. On the other hand, repositories that get a large number of pull requests tend to be quite large themselves and have support from a good number of developers, making it easier for them to evaluate pull requests submitted against them. It is also possible that many of them are very open to accepting contributions, which is known to many of the PR creators as well, which makes them more willing to submit pull requests to these repositories.

5.6.4.6 Variation with the total number of projects a PR creator contributed to

We observe that having contributed to a larger number of projects steadily increase the chance of a PR creator's contributions being accepted, which is likely because developers who have contributed to a larger number of projects tend to be more experienced, more knowledgeable about the different requirements of different projects, and, in general, tend to create better pull requests. However, a trend reversal is observed for creators who contributed to over 5000 projects. The reason for the trend reversal wasn't clear to us at first, so we investigated those cases further, and found that these developers tend to be bots (e.g. greenkeeper bot), not humans, which explains why the pull requests created by them have a relatively lower chance of being accepted, since these bots do not gain experience and improve in the same way as human developers.

5.6.4.7 Variation with the total number of commits by a PR creator

The probability of a pull request being accepted tends to increase with the total number of commits created by a PR creator, however, we observe a trend reversal for PR creators with an extremely large number of commits. The reason for an increase in PR acceptance probability with a larger number of commits is quite straightforward: the PR creator is more experienced, so the code contributions they make tend to be of higher quality. The reason for the trend reversal for creators with a very high number of commits (over 10,000) is, once again, because most of them actually are bots.

5.6.4.8 Variation with the number of commits in the pull request

We observe from the partial dependence plot that pull requests with very few commits have a low chance of getting accepted, since those might not have a significant amount of contribution. Initially, the probability of PR acceptance increases rapidly as the number of commits increase. However, after reaching a peak at around 2 commits, the probability of acceptance starts dropping quickly. The drop rate slows down for pull requests with over 10 commits and gets saturated for the ones with over 300 commits.

5.6.4.9 Variation with the number of lines added in the pull request

Similar to what we observe for the commits, the pull requests with very few lines added are less likely to be accepted. The probability rises to a peak for the pull requests with around 20 lines added, starts dropping steadily until we reach pull requests with around 400 lines added, and the rate of decrease in the chance of a PR being accepted keeps dropping slowly after that.

5.6.4.10 Variation with the number of code review comments for the pull request

The partial dependence plot shows that the probability of acceptance is high for the pull requests with no code review comments, the value of the acceptance probability takes a plunge for the ones with just a single code review comment (likely the comment clarifies why it couldn't be accepted), and shows a continuous moderate increase for the ones with more than 1 code review comments. This potentially reflects the fact that some PRs may require more discussion due to their complexity or impact, but are otherwise of high quality.

5.6.4.11 Variation with the number of changed files in the pull request

As with the other measures related to the size of the PR, we see a steady increase in the probability of PR acceptance for the pull requests with up to 4 changed files, and it shows a constant decrease after that. It is worth mentioning that most ($\sim 80\%$) of the pull requests in our dataset had less than 4 changed files, which is likely why we saw a positive regression coefficient for this variable in our logistics regression model (Table 5-2).

5.6.4.12 Variation with the number of pull requests submitted by the PR creator

The variation of the PR acceptance probability with the number of pull requests submitted by a PR creator is a bit complex, and we have to keep in mind that our dataset has multiple entries corresponding to a PR creator, one for each of their submitted pull requests, to fully comprehend the dependence. The initial peak in the dependence plot most likely corresponds to the "cold start" problem, i.e. when the PR creators with a high amount of skill submit their pull requests for the first time and it gets accepted. The following trough reaches its lowest point at around 10 pull requests submitted by the PR creator, and then we see the experience gained by the PR creators adding up, increasing the probability of their pull requests being accepted. The probability reaches its peak at around 100 pull requests by a PR creator, and maintains its value until reaching around 1200 pull requests. The only PR creators who create more pull requests are almost exclusively bots (e.g. greenkeeper bot), and they tend to create pull requests that do not really reflect the experience gained by a human developer, and have a much lower chance of being accepted.

5.6.4.13 Dependence on previous social proximity , containing an issue fix, and presence of a dependency

We do not have a variation of the PR acceptance probability per se for the three categorical variables, but the partial dependence plots show that having a previous PR accepted in the repository, a dependence between the package and the projects the PR creator contributed to, and an explicit mention of an issue fix tend to have a positive impact on the PR acceptance probability.

Answering RQ3: The variation of the PR acceptance probability and the measures we presented in this study are depicted using the partial dependence plots (Figure 5-3) and the likely cause for such variation is discussed in detail. Overall, we observe that the nature of the variation is often nonlinear with peaks and troughs, something that can't be captured adequately by simple regression models, which was used by most of the previous studies to describe the nature of the relationships

5.7 Limitations

In our study, we focused only on the more popular NPM packages, which constitute less than 0.5% of the entire NPM ecosystem. Although these are the packages that see the most amount of activity, and should, therefore, be of interest to most of the practitioners and researchers in the field, some of the factors found to significant for these packages might not be so important in the pull request acceptance scenario for the less popular packages.

The result of this study might not be applicable as-is to other software ecosystems, since every ecosystem has their norms and characteristics which is impossible to account for when looking into only one ecosystem. Future studies are needed to determine the generality of our findings.

Some of the characteristics (e.g. see Section 5.6.4) observed in our study could be due to the presence of bots in the dataset that behave differently from human developers. Being able to get rid of them would further improve the accuracy of our findings, which we plan to do as a future work. Another important topic that could be addressed by future work is finding out if the PR integrators actually find these signals to be useful and identify any factors we might have missed here.

5.8 Summary: Addressing RT4

The goal of this study was to address the fourth research topic mentioned in subsection 1.2.1: *"Investigating the effects of the characteristics of individual patch contributors, specifically, their experience and social proximity to the repositories to which they submit PRs, on the chances of their pull requests getting accepted."*

Therefore, in this study, we aimed to examine the effects of various social and technical factors on the quality of a pull request (its probability of being accepted). We formed seven hypotheses that replicate findings in prior work and also pose additional propositions that reflect the ecosystem-wide concerns. We fit logistic regression models that show statistically significant relationship of PR acceptance and 15 hypothesised predictors. We also predict the acceptance of future PRs with AUC-ROC of 0.94. Finally we explore the functional relationship between the predictors and the probability of pull request acceptance and find it to be nonlinear and even non-monotone in many cases.

Our findings provide evidence to fact that the characteristics of a user, especially their overall experience, their track record of having their PRs accepted, and their social connection with the repository to which they are submitting a PR, have strong influence on the chance of their PRs being accepted. The social connection factor once again highlights the importance of adopting a supply chain perspective for analyzing similar questions.

Our findings have both theoretical and practical implications. The accuracy of our PR acceptance model increases the likelihood of successful practical applications that range from tools that support PR integrators to tools that help the PR creators to tailor their contributions to the form resembling that of the pull requests that are most likely to be accepted by a specific project. We plan to pursue the goal of evaluating such tools in OSS projects. As the NPM ecosystem and other OSS ecosystems depend on contributors to maintain growth and code quality, we hope that the results of our work would help these ecosystems to sustain evolution and the high quality of the code.

CHAPTER 6
DISCUSSION AND CONCLUSION

6.1 Goal of the Research

The primary goal of the research was to model various properties that are affected by users of a software and their interaction with it, as well as by the interconnected nature of the software ecosystems. We have identified a number of such properties (see Figure 1-2), viz. the quality of the software from the users' perspective, the popularity of a software, the issues and PRs created against a software package and how they depend on the issue/ PR creators connection with the package. We also found that the presence of bots in the datasets to be a serious problem, which can lead to misleading conclusions due to the nature of bots being different from human developers, so we worked on developing a systematic method for detecting bots in order to be able to clean the datasets.

6.2 Key Findings

In the research, we have explored the four mail research topics listed in subsection 1.2.1 (RT1-RT4) as four different projects, which are described in detail in Chapters 2-5. The key findings related to the research topics are listed below:

RT1: The usage of a software was found to have a strong influence on the number of software faults, which can be thought of as a measure of the quality of the software from the users' perspective, even after accounting for the code complexity measures. The result holds even when using inexact measures for software faults or usage.

RT2: For the NPM ecosystem, the number of dependents of a package and their popularity could almost entirely explain the popularity of a software package, and were important predictors for predicting whether the popularity of a package will increase or decrease. The effects of recursive (transitive) dependencies and dependents were also found to equally important, showing the benefit of adopting a supply chain view of the software ecosystems.

RT3: The results showed that a lot of issues and PRs to various projects are in fact created by developers who do directly depend on the packages, which shows the importance of the supply chain perspective into the ecosystem. At the same time, we found a possible issue with the aspect of visibility, at least for the NPM ecosystem, and a number of users were found to create issues against packages they have no visible connection with, which warrants further investigation.

RT4: Our findings provide evidence to fact that the characteristics of a user, especially their overall experience, their track record of having their PRs accepted, and their social connection with the repository to which they are submitting a PR, have strong influence on the chance of their PRs being accepted. The social connection factor once again highlights the importance of adopting a supply chain perspective for analyzing similar questions.

In addition, we took up the task of detecting bots, which would be useful for cleaning the dataset and improving the accuracy of the results, and designed a systematic method for detecting code-commit bots called **BIMAN**, which gave an AUC-ROC value of 0.94 while trying to find bots. The result of this project is presented in Appendix A.

6.3 Discussion

The findings of the research sheds some light on the bigger picture of identifying how various factors that have an influence over the ultimate success of a software (according to the Information System Success Model) are related to each other, and are affected by the users of a software and the interconnected nature of the software ecosystems.

The interconnected nature of a software ecosystem was found to have a big influence over the popularity of the packages in the ecosystem, as well as on which packages an external contributor submits issues against, and how likely their PRs get accepted by those packages. As shown by our proposed analogue (see Figure 1-2) of corresponding properties in an Information System and by the IS success model (see Figure 1-1), these properties influence the ultimate success of a software. Therefore, the results of this research *establishes the relationship between the success of a software and the interdependence between the software modules in the corresponding ecosystem.* Adopting a supply chain perspective was found to be invaluable in analyzing the interconnection between the software modules, so, we recommend using that framework for similar studies based on our experience.

Our research also explored the effect the users of a software have on its properties. The results of the study exploring RT1 (see Chapter 2) established the relationship between the aspects of “System Use/ Usage Intention” and “User Satisfaction” (see the analogue in Figure 1-2), which was also shown in the IS success model in Figure 1-1. Our research also modeled the influence of the users’ characteristics on Issues and PRs, which are important measures of “User Statisfaction”, and a mode of communication between the external contributors and the core developers, an aspect missing in the original IS success model. The results of the research shows how important it is to account for the users’ characteristics while studying these properties, and, by extension, their importance in determining the ultimate success of a software.

6.4 Conclusion

The ultimate success of a software is a complex, and somewhat illusive topic, which has nontrivial interaction with various other properties of the software, as well as that of the core developers, the contributors, and other factors ranging from the capability of the hardware supporting it, the cognitive biases and limitations of humans, to the social and geopolitical situation of the period. The results of this research takes a small step towards understanding this complex situation, and highlighted some of the important interactions by using the conceptual frameworks of IS success model and Software Supply Chain.

There are a significant number of ways to work towards in future, including testing the validity of the results for other ecosystems, addressing the limitations of the individual projects, as listed in the corresponding chapters, and studying the aspects that were left out in this research. Due to the presence of a number of ways the external contributors can communicate with the core developers, an aspect missing in the original IS success model, it would be interesting to observe and quantify the impact of external contributors on the “internal” properties of the software system.

LIST OF REFERENCES

[1] IEEE, “Ieee standard glossary of software engineering terminology,” *IEEE Std 610.12-1990*, pp. 1–84, 1990.

[2] M. F. Lungu, *Reverse engineering software ecosystems*. PhD thesis, Università della Svizzera italiana, 2009.

[3] N. Eghbal, *Roads and bridges: The unseen labor behind our digital infrastructure*. Ford Foundation, 2016.

[4] G. Piccoli and F. Pigni, *Information Systems for Managers*. John Wiley & Sons, 2008.

[5] W. H. DeLone and E. R. McLean, “Information systems success: The quest for the dependent variable,” *Information systems research*, vol. 3, no. 1, pp. 60–95, 1992.

[6] S. Peters, “The role of users in open source projects.” O’Reilly Oscon Open Source Convention, 2009.

[7] D. M. Nichols and M. B. Twidale, “Usability processes in open source projects,” *Software Process: Improvement and Practice*, vol. 11, no. 2, pp. 149–162, 2006.

[8] L. Damodaran, “User involvement in the systems design process-a practical guide for users,” *Behaviour & information technology*, vol. 15, no. 6, pp. 363–377, 1996.

[9] N. Ducheneaut, “Socialization in an open source software community: A socio-technical analysis,” *Computer Supported Cooperative Work (CSCW)*, vol. 14, no. 4, pp. 323–368, 2005.

[10] R. P. Bagozzi and U. M. Dholakia, “Open source software user communities: A study of participation in linux user groups,” *Management science*, vol. 52, no. 7, pp. 1099–1115, 2006.

[11] K. R. Lakhani and E. Von Hippel, “How open source software works:“free” user-to-user assistance,” in *Produktentwicklung mit virtuellen Communities*, pp. 303–339, Springer, 2004.

[12] A. Guzzi, A. Bacchelli, M. Lanza, M. Pinzger, and A. v. Deursen, “Communication in open source software development mailing lists,” in *Proceedings of the 10th Working Conference on Mining Software Repositories*, pp. 277–286, IEEE Press, 2013.

[13] E. Horvitz, J. Breese, D. Heckerman, D. Hovel, and K. Rommelse, "The lumiere project: Bayesian user modeling for inferring the goals and needs of software users," in *Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence*, pp. 256–265, Morgan Kaufmann Publishers Inc., 1998.

[14] G. H. Walton, J. H. Poore, and C. J. Trammell, "Statistical testing of software based on a usage model," *Software: Practice and Experience*, vol. 25, no. 1, pp. 97–108, 1995.

[15] S. S. Lew, P. Kim, S. Katariya, and Z. Zheng, "Adaptive systems and methods for making software easy to use via software usage mining," Sept. 21 2010. US Patent 7,802,197.

[16] M. El-Ramly and E. Stroulia, "Mining software usage data," in *Proceedings of 1st International Workshop on Mining Software Repositories (MSR'04)*, pp. 64–8, 2004.

[17] R. Girardi, L. B. Marinho, and I. R. De Oliveira, "A system of agent-based software patterns for user modeling based on usage mining," *Interacting with computers*, vol. 17, no. 5, pp. 567–591, 2005.

[18] A. G. Oettinger, "President's letter to the acm membership," *Commun. ACM*, vol. 9, p. 545–546, Aug. 1966.

[19] R. K. Oliver, M. D. Webber, *et al.*, "Supply-chain management: logistics catches up with strategy," *Outlook*, vol. 5, no. 1, pp. 42–47, 1982.

[20] S. Amreen, B. Bichescu, R. Bradley, T. Dey, Y. Ma, A. Mockus, S. Mousavi, and R. Zaretzki, "A methodology for measuring floss ecosystems," in *Towards Engineering Free/Libre Open Source Software (FLOSS) Ecosystems for Impact and Sustainability*, pp. 1–29, Springer, Singapore, 2019.

[21] J. Holdsworth, *Software Process Design.* McGraw-Hill, Inc., 1995.

[22] B. Farbey and A. Finkelstein, "Exploiting software supply chain business architecture: a research agenda," 1999.

[23] J. Greenfield and K. Short, "Software factories: assembling applications with patterns, models, frameworks and tools," in *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 16–27, ACM, 2003.

[24] E. Levy, "Poisoning the software supply chain," *Security & Privacy, IEEE*, vol. 1, no. 3, pp. 70–73, 2003.

[25] A. A. Chhajed and S. H. Xu, "Software focused supply chains: Challenges and issues," in *Industrial Informatics, 2005. INDIN'05. 2005 3rd IEEE International Conference on*, pp. 172–175, IEEE, 2005.

[26] T. Dey and A. Mockus, "Modeling relationship between post-release faults and usage in mobile software," in *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE'18, (New York, NY, USA), pp. 56–65, Association for Computing Machinery, 2018.

[27] T. Dey and A. Mockus, "Deriving a usage-independent software quality metric," *Empirical Software Engineering*, vol. 25, no. 2, pp. 1596–1641, 2020.

[28] A. Mockus and D. Weiss, "Interval quality: Relating customer-perceived quality to process quality," in *2008 International Conference on Software Engineering*, (Leipzig, Germany), pp. 733–740, ACM Press, May 10–18 2008.

[29] R. Hackbarth, A. Mockus, J. Palframan, and R. Sethi, "Customer quality improvement of software systems," *Software, IEEE*, vol. 33, no. 4, pp. 40–45, 2016.

[30] Y. Ma, C. Bogart, S. Amreen, R. Zaretzki, and A. Mockus, "World of code: An infrastructure for mining the universe of open source vcs data," in *IEEE Working Conference on Mining Software Repositories*, May 2019.

[31] T. Dey and A. Mockus, "Are software dependency supply chain metrics useful in predicting change of popularity of npm packages?," in *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE'18, (New York, NY, USA), pp. 66–69, Association for Computing Machinery, 2018.

[32] T. Dey, Y. Ma, and A. Mockus, "Patterns of effort contribution and demand and user classification based on participation patterns in npm ecosystem," in *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE'19, (New York, NY, USA), pp. 36–45, Association for Computing Machinery, 2019.

[33] T. Dey and A. Mockus, "Effect of technical and social factors on pull request quality for the npm ecosystem," *arXiv preprint arXiv:2007.04816*, 2020.

[34] T. Dey, S. Mousavi, E. Ponce, T. Fry, B. Vasilescu, A. Filippova, and A. Mockus, "Detecting and characterizing bots that commit code," *arXiv preprint arXiv:2003.03172*, 2020.

[35] N. Fenton, M. Neil, and D. Marquez, "Using bayesian networks to predict software defects and reliability," *Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability*, vol. 222, no. 4, pp. 701–712, 2008.

[36] N. E. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Transactions on software engineering*, vol. 25, no. 5, pp. 675–689, 1999.

[37] C. Jones, "Software quality in 2011: A survey of the state of the art." http://sqgne.org/presentations/2011-12/Jones-Sep-2011.pdf, 2011. President, Namcook Analytics LLC, www.Namcook.com Email: Capers.Jones3@GMAILcom.

[38] A. Krutauz, T. Dey, P. C. Rigby, and A. Mockus, "Do code review measures explain the incidence of post-release defects?," *arXiv preprint arXiv:2005.09217*, 2020.

[39] S. Mcintosh, Y. Kamei, B. Adams, and A. E. Hassan, "An empirical study of the impact of modern code review practices on software quality," *Empirical Softw. Engg.*, vol. 21, pp. 2146–2189, Oct. 2016.

[40] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, pp. 192–201, ACM, 2014.

[41] P. C. Rigby and C. Bird, "Convergent contemporary software peer review practices," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pp. 202–212, ACM, 2013.

[42] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. W. Godfrey, "Investigating code review quality: Do people and participation matter?," in *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pp. 111–120, IEEE, 2015.

[43] K. C. Chatzidimitriou, M. D. Papamichail, T. Diamantopoulos, M. Tsapanos, and A. L. Symeonidis, "npm-miner: An infrastructure for measuring the quality of the npm registry," in *Proceedings of the 15th International Conference on Mining Software Repositories*, pp. 42–45, ACM, 2018.

[44] A. N. Duc, A. Mockus, R. Hackbarth, and J. Palframan, "Forking and coordination in multi-platform development: a case study," in *ESEM*, (Torino, Italy), pp. 59:1–59:10, September 2014.

[45] L. Voss, "numeric precision matters: how npm download counts work," 2014.

[46] David, 2014.

[47] L. Voss, "The state of javascript frameworks, 2017," 2018.

[48] H. Borges, A. Hora, and M. T. Valente, "Understanding the factors that impact the popularity of github repositories," in *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*, pp. 334–344, IEEE, 2016.

[49] D. Koller and N. Friedman, *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[50] M. Scutari and K. Strimmer, "Introduction to graphical modelling," *arXiv preprint arXiv:1005.1036*, 2010.

[51] P. Yu, T. Systa, and H. Muller, "Predicting fault-proneness using oo metrics. an industrial case study," in *Software Maintenance and Reengineering, 2002. Proceedings. Sixth European Conference on*, pp. 99–107, IEEE, 2002.

[52] R. Subramanyam and M. S. Krishnan, "Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects," *IEEE Transactions on software engineering*, vol. 29, no. 4, pp. 297–310, 2003.

[53] L. C. Briand, J. Wüst, J. W. Daly, and D. V. Porter, "Exploring the relationships between design measures and software quality in object-oriented systems," *Journal of systems and software*, vol. 51, no. 3, pp. 245–273, 2000.

[54] A. Mockus, "Software support tools and experimental work," in *Empirical Software Engineering Issues: Critical Assessments and Future Directions* (V. Basili and et al, eds.), vol. LNCS 4336, pp. 91–99, Springer, 2007.

[55] A. Mockus, "Engineering big data solutions," in *ICSE'14 FOSE*, pp. 85–99, 2014.

[56] Q. Zheng, A. Mockus, and M. Zhou, "A method to identify and correct problematic software activity data: Exploiting capacity constraints and data redundancies," in *ESEC/FSE'15*, (Bergamo, Italy), pp. 637–648, ACM, 2015.

[57] D. M. Chickering, “Learning bayesian networks is np-complete,” *Learning from data: Artificial intelligence and statistics V*, vol. 112, pp. 121–130, 1996.

[58] M. Scutari, “Learning bayesian networks in r, an example in systems biology,” 2013. http://www.bnlearn.com/about/slides/slides-useRconf13.pdf.

[59] R. Nagarajan, M. Scutari, and S. Lèbre, “Bayesian networks in r,” *Springer*, vol. 122, pp. 125–127, 2013.

[60] S. G. Bottcher and C. Dethlefsen., *deal: Learning Bayesian Networks with Mixed Variables*, 2013. R package version 1.2-37.

[61] N. Balov and P. Salzman, *catnet: Categorical Bayesian Network Inference*, 2016. R package version 1.15.0.

[62] Alain Hauser, Peter Buehlmann, “Characterization and greedy learning of interventional markov equivalence classes of directed acyclic graphs,” *Journal of Machine Learning Research,*, vol. 13, pp. 2409–2464, 2012.

[63] M. Kalisch, M. Mächler, D. Colombo, M. H. Maathuis, and P. Bühlmann, “Causal inference using graphical models with the R package pcalg,” *Journal of Statistical Software*, vol. 47, no. 11, pp. 1–26, 2012.

[64] G. Shmueli, “To explain or to predict?,” *Statistical science*, pp. 289–310, 2010.

[65] E. Sober, “Instrumentalism, parsimony, and the akaike framework,” *Philosophy of Science*, vol. 69, no. S3, pp. S112–S123, 2002.

[66] N. Friedman, M. Goldszmidt, and A. Wyner, “Data analysis with bayesian networks: A bootstrap approach,” in *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*, pp. 196–205, Morgan Kaufmann Publishers Inc., 1999.

[67] B. S. Chlebus and S. H. Nguyen, “On finding optimal discretizations for two attributes,” in *International Conference on Rough Sets and Current Trends in Computing*, pp. 537–544, Springer, 1998.

[68] A. Perez, P. Larranaga, and I. Inza, “Supervised classification with conditional gaussian networks: Increasing the structure complexity from naive bayes,” *International Journal of Approximate Reasoning*, vol. 43, no. 1, pp. 1–25, 2006.

[69] M. Hahsler, S. Chelluboina, K. Hornik, and C. Buchta, "The arules r-package ecosystem: Analyzing interesting patterns from large transaction datasets," *Journal of Machine Learning Research*, vol. 12, pp. 1977–1981, 2011.

[70] A. J. Hartemink, *Principled computational methods for the validation and discovery of genetic regulatory networks*. PhD thesis, Massachusetts Institute of Technology, 2001.

[71] Marco Scutari, "Learning bayesian networks with the bnlearn r package.," *Journal of Statistical Software*, vol. 35, no. 3, pp. 1–22, 2010.

[72] J. Pearl, "Bayesian networks," *Department of Statistics, UCLA*, 2011.

[73] A. Mockus, "Law of minor release: More bugs implies better software quality." http://mockus.org/papers/IWPSE13.pdf, 2013. International Workshop on Principles of Software Evolution, St Petersburg, Russia, Aug 18-19 2013. Keynote.

[74] H. M. (https://stats.stackexchange.com/users/25/harvey motulsky), "When is r squared negative?." Cross Validated. URL:https://stats.stackexchange.com/q/12991 (version: 2014-05-06).

[75] A. Mockus, P. Zhang, and P. L. Li, "Predictors of customer perceived software quality," in *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pp. 225–233, IEEE, 2005.

[76] A. Mockus, P. Zhang, and P. Li, "Drivers for customer perceived software quality," in *ICSE 2005*, (St Louis, Missouri), pp. 225–233, ACM Press, May 2005.

[77] A. Mockus and D. Weiss, "Interval quality: Relating customer-perceived quality to process quality," in *Proceedings of the 30th international conference on Software engineering*, pp. 723–732, ACM, 2008.

[78] B. W. Boehm, J. R. Brown, and M. Lipow, "Quantitative evaluation of software quality," in *Proceedings of the 2nd international conference on Software engineering*, pp. 592–605, IEEE Computer Society Press, 1976.

[79] B. Kitchenham and S. L. Pfleeger, "Software quality: the elusive target [special issues section]," *IEEE software*, vol. 13, no. 1, pp. 12–21, 1996.

[80] J. Rubin and M. Rinard, "The challenges of staying together while moving fast: An exploratory study," in *Proceedings of the 38th International Conference on Software Engineering*, pp. 982–993, ACM, 2016.

[81] S. R. Dalal and C. L. Mallows, "When should one stop testing software?," *Journal of the American Statistical Association*, vol. 83, no. 403, pp. 872–879, 1988.

[82] A. Mockus, R. Hackbarth, and J. Palframan, "Risky files: An approach to focus quality improvement effort," in *9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 691–694, 2013.

[83] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou, "Towards building a universal defect prediction model with rank transformed predictors," *Empirical Software Engineering*, pp. 1–39, 2015.

[84] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2013.

[85] A. Mockus and D. M. Weiss, "Predicting risk of software changes," *Bell Labs Technical Journal*, vol. 5, pp. 169–180, April–June 2000.

[86] P. L. Li, R. Kivett, Z. Zhan, S.-e. Jeon, N. Nagappan, B. Murphy, and A. J. Ko, "Characterizing the differences between pre-and post-release versions of software," in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 716–725, ACM, 2011.

[87] G. Q. Kenny, "Estimating defects in commercial software during operational use," *IEEE Transactions on Reliability*, vol. 42, no. 1, pp. 107–115, 1993.

[88] P. Rotella and S. Chulani, "Implementing quality metrics and goals at the corporate level," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, pp. 113–122, ACM, 2011.

[89] T. Geron, "Do ios apps crash more than android apps? a data dive," 2012. https://www.forbes.com/sites/tomiogeron/2012/02/02/does-ios-crash-more-than-android-a-data-dive.

[90] F. Khomh, T. Dhaliwal, Y. Zou, and B. Adams, "Do faster releases improve software quality?: an empirical case study of mozilla firefox," in *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pp. 179–188, IEEE Press, 2012.

[91] N. Fenton, P. Krause, and M. Neil, "Software measurement: Uncertainty and causal modeling," *IEEE software*, vol. 19, no. 4, pp. 116–122, 2002.

[92] J. D. Herbsleb and A. Mockus, "An empirical study of speed and communication in globally-distributed software development," *IEEE Transactions on Software Engineering*, vol. 29, pp. 481–494, June 2003.

[93] I. Stamelos, L. Angelis, P. Dimou, and E. Sakellaris, "On the use of bayesian belief networks for the prediction of software productivity," *Information and Software Technology*, vol. 45, no. 1, pp. 51–60, 2003.

[94] P. C. Pendharkar, G. H. Subramanian, and J. A. Rodger, "A probabilistic model for predicting software development effort," *IEEE Transactions on software engineering*, vol. 31, no. 7, pp. 615–624, 2005.

[95] N. Fenton, M. Neil, W. Marsh, P. Hearty, D. Marquez, P. Krause, and R. Mishra, "Predicting software defects in varying development lifecycles using bayesian nets," *Information and Software Technology*, vol. 49, no. 1, pp. 32–43, 2007.

[96] M. Neil and N. Fenton, "Predicting software quality using bayesian belief networks," in *Proceedings of the 21st Annual Software Engineering Workshop*, pp. 217–230, NASA Goddard Space Flight Centre, 1996.

[97] A. Okutan and O. T. Yıldız, "Software defect prediction using bayesian networks," *Empirical Software Engineering*, vol. 19, no. 1, pp. 154–181, 2014.

[98] G. J. Pai and J. B. Dugan, "Empirical analysis of software fault content and fault proneness using bayesian methods," *IEEE Transactions on software Engineering*, vol. 33, no. 10, pp. 675–686, 2007.

[99] R. Hackbarth, A. Mockus, J. Palframan, and R. Sethi, "Improving software quality as customers perceive it," *IEEE Software*, vol. 33, no. 4, pp. 40–45, 2016.

[100] S. H. Kan, *Metrics and models in software quality engineering*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[101] G. G. Schulmeyer and J. I. McManus, *Handbook of software quality assurance*. Van Nostrand Reinhold Co., 1992.

[102] E. Wittern, P. Suter, and S. Rajagopalan, "A look at the dynamics of the javascript package ecosystem," in *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*, pp. 351–361, IEEE, 2016.

[103] A. Zerouali, E. Constantinou, T. Mens, G. Robles, and J. González-Barahona, “An empirical analysis of technical lag in npm package dependencies,” in *International Conference on Software Reuse*, pp. 95–110, Springer, 2018.

[104] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab, “Why do developers use trivial packages? an empirical case study on npm,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 385–395, ACM, 2017.

[105] D. Datta and S. Kajanan, “Do app launch times impact their subsequent commercial success? an analytical approach,” in *2013 International Conference on Cloud Computing and Big Data (CloudCom-Asia)*, pp. 205–210, IEEE, 2013.

[106] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, “Api change and fault proneness: a threat to the success of android apps,” in *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, pp. 477–487, ACM, 2013.

[107] I. J. M. Ruiz, M. Nagappan, B. Adams, T. Berger, S. Dienst, and A. E. Hassan, “Impact of ad libraries on ratings of android mobile apps,” *IEEE Software*, vol. 31, no. 6, pp. 86–92, 2014.

[108] Y. Tian, M. Nagappan, D. Lo, and A. E. Hassan, “What are the characteristics of high-rated apps? a case study on free android applications,” in *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pp. 301–310, IEEE, 2015.

[109] L. Guerrouj, S. Azad, and P. C. Rigby, “The influence of app churn on app success and stackoverflow discussions,” in *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pp. 321–330, IEEE, 2015.

[110] S. Weber and J. Luo, “What makes an open source code popular on git hub?,” in *Data Mining Workshop (ICDMW), 2014 IEEE International Conference on*, pp. 851–855, IEEE, 2014.

[111] J. Zhu, M. Zhou, and A. Mockus, “Patterns of folder use and project popularity: A case study of github repositories,” in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, p. 30, ACM, 2014.

[112] A. Zerouali, T. Mens, G. Robles, and J. M. Gonzalez-Barahona, “On the diversity of software package popularity metrics: An empirical study of npm,” in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 589–593, IEEE, 2019.

[113] S. Wood, *Generalized Additive Models: An Introduction with R.* Chapman and Hall/CRC, 2 ed., 2017.

[114] R. L. Young, J. Weinberg, V. Vieira, A. Ozonoff, and T. F. Webster, “Generalized additive models and inflated type i error rates of smoother significance tests,” *Computational statistics & data analysis*, vol. 55, no. 1, pp. 366–374, 2011.

[115] S. N. Wood, “On p-values for smooth components of an extended generalized additive model,” *Biometrika*, vol. 100, no. 1, pp. 221–228, 2012.

[116] E. Von Hippel, “Learning from open-source software,” *MIT Sloan management review*, vol. 42, no. 4, pp. 82–86, 2001.

[117] J. Xie, M. Zhou, and A. Mockus, “Impact of triage: a study of mozilla and gnome,” in *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 247–250, IEEE, 2013.

[118] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, “How to break an api: Cost negotiation and community values in three software ecosystems,” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 109–120, ACM, 2016.

[119] M. Valiev, B. Vasilescu, and J. Herbsleb, “Ecosystem-level determinants of sustained activity in open-source projects: a case study of the pypi ecosystem,” in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 644–655, ACM, 2018.

[120] B. W. Boehm, “Software risk management: principles and practices,” *IEEE software*, vol. 8, no. 1, pp. 32–41, 1991.

[121] L. Wallace, M. Keil, and A. Rai, “Understanding software project risk: a cluster analysis,” *Information & management*, vol. 42, no. 1, pp. 115–125, 2004.

[122] P. C. Rigby, Y. C. Zhu, S. M. Donadelli, and A. Mockus, “Quantifying and mitigating turnover-induced knowledge loss: case studies of chrome and a project at avaya,” in

2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), pp. 1006–1016, IEEE, 2016.

[123] C. J. Alberts, A. J. Dorofee, R. Creel, R. J. Ellison, and C. Woody, "A systemic approach for assessing software supply-chain risk," in *2011 44th Hawaii International Conference on System Sciences*, pp. 1–8, IEEE, 2011.

[124] A. Lee, J. C. Carver, and A. Bosu, "Understanding the impressions, motivations, and barriers of one time code contributors to floss projects: a survey," in *Proceedings of the 39th International Conference on Software Engineering*, pp. 187–197, IEEE Press, 2017.

[125] A. Lee and J. C. Carver, "Are one-time contributors different? a comparison to core and periphery developers in floss repositories," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 1–10, IEEE, 2017.

[126] M. Zhou, A. Mockus, X. Ma, L. Zhang, and H. Mei, "Inflow and retention in oss communities with commercial involvement: A case study of three hybrid projects," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 2, p. 13, 2016.

[127] L. Voss, "how many npm users are there?," 2016.

[128] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pp. 181–191, IEEE, 2018.

[129] R. E. Zapata, R. G. Kula, B. Chinthanet, T. Ishio, K. Matsumoto, and A. Ihara, "Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm javascript packages," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 559–563, IEEE, 2018.

[130] A. Decan, T. Mens, and M. Claes, "An empirical comparison of dependency issues in oss packaging ecosystems," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 2–12, IEEE, 2017.

[131] A. Decan, T. Mens, M. Claes, and P. Grosjean, "When github meets cran: An analysis of inter-repository package dependency problems," in *2016 IEEE 23rd*

International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol. 1, pp. 493–504, IEEE, 2016.

[132] G. Canfora, L. Cerulo, M. Cimitile, and M. Di Penta, "Social interactions around cross-system bug fixings: the case of freebsd and openbsd," in *Proceedings of the 8th working conference on mining software repositories*, pp. 143–152, ACM, 2011.

[133] W. Ma, L. Chen, X. Zhang, Y. Zhou, and B. Xu, "How do developers fix cross-project correlated bugs? a case study on the github scientific python ecosystem," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 381–392, IEEE, 2017.

[134] H. Ding, W. Ma, L. Chen, Y. Zhou, and B. Xu, "An empirical study on downstream workarounds for cross-project bugs," in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 318–327, IEEE, 2017.

[135] A. Mockus, R. T. Fielding, and J. D. Herbsleb, "Two case studies of open source software development: Apache and mozilla," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, no. 3, pp. 309–346, 2002.

[136] E. Glynn, B. Fitzgerald, and C. Exton, "Commercial adoption of open source software: an empirical study," in *2005 International Symposium on Empirical Software Engineering, 2005.*, pp. 10–pp, IEEE, 2005.

[137] P. Y. Chau and K. Y. Tam, "Factors affecting the adoption of open systems: an exploratory study," *MIS quarterly*, pp. 1–24, 1997.

[138] M. Ciesielska and A. Westenholz, "Dilemmas within commercial involvement in open source software," *Journal of Organizational Change Management*, vol. 29, no. 3, pp. 344–360, 2016.

[139] J. C. Bezdek, R. Ehrlich, and W. Full, "Fcm: The fuzzy c-means clustering algorithm," *Computers & Geosciences*, vol. 10, no. 2-3, pp. 191–203, 1984.

[140] G. Gousios, "The ghtorrent dataset and tool suite," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, (Piscataway, NJ, USA), pp. 233–236, IEEE Press, 2013.

[141] K. Crowston and J. Howison, "The social structure of open source software development teams," 2003.

[142] P. Wagstrom, C. Jergensen, and A. Sarma, “Roles in a networked software development ecosystem: A case study in github,” 2012.

[143] L. Dabbish, C. Stuart, J. Tsay, and J. 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*, pp. 1277–1286, ACM, 2012.

[144] J. Zhu, M. Zhou, and A. Mockus, “Effectiveness of code contribution: From patch-based to pull-request-based tools,” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 871–882, ACM, 2016.

[145] G. Gousios, A. Zaidman, M.-A. Storey, and A. 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*, pp. 358–368, IEEE Press, 2015.

[146] P. C. Rigby, D. M. German, L. Cowen, and M.-A. Storey, “Peer review on open-source software projects: Parameters, statistical models, and theory,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 4, p. 35, 2014.

[147] P. Weißgerber, D. Neu, and S. Diehl, “Small patches get in!,” in *Proceedings of the 2008 international working conference on Mining software repositories*, pp. 67–76, ACM, 2008.

[148] D. M. Soares, M. L. de Lima Júnior, L. Murta, and A. Plastino, “Acceptance factors of pull requests in open-source projects,” in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pp. 1541–1546, ACM, 2015.

[149] M. M. Rahman and C. K. Roy, “An insight into the pull requests of github,” in *Proceedings of the 11th Working Conference on Mining Software Repositories*, pp. 364–367, ACM, 2014.

[150] Y. Jiang, B. Adams, and D. 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*, pp. 101–110, IEEE Press, 2013.

[151] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey, “The secret life of patches: A firefox case study,” in *2012 19th Working Conference on Reverse Engineering*, pp. 447–455, IEEE, 2012.

[152] T. Dey and A. Mockus, “A Dataset of Pull Requests and A Trained Random Forest Model for predicting Pull Request Acceptance,” May 2020.

[153] C. Maddila, C. Bansal, and N. Nagappan, “Predicting pull request completion time: a case study on large scale cloud services,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 874–882, 2019.

[154] Y. Yu, H. Wang, G. Yin, and C. X. Ling, “Who should review this pull-request: Reviewer recommendation to expedite crowd collaboration,” in *2014 21st Asia-Pacific Software Engineering Conference*, vol. 1, pp. 335–342, IEEE, 2014.

[155] Y. Yu, H. Wang, G. Yin, and C. X. Ling, “Reviewer recommender of pull-requests in github,” in *2014 IEEE International Conference on Software Maintenance and Evolution*, pp. 609–612, IEEE, 2014.

[156] J. Jiang, Y. Yang, J. He, X. Blanc, and L. Zhang, “Who should comment on this pull request? analyzing attributes for more accurate commenter recommendation in pull-based development,” *Information and Software Technology*, vol. 84, pp. 48–62, 2017.

[157] Y. Yu, H. Wang, G. Yin, and T. Wang, “Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment?,” *Information and Software Technology*, vol. 74, pp. 204–218, 2016.

[158] E. v. d. Veen, G. Gousios, and A. Zaidman, “Automatically prioritizing pull requests,” in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pp. 357–361, May 2015.

[159] G. Gousios, M.-A. Storey, and A. Bacchelli, “Work practices and challenges in pull-based development: the contributor’s perspective,” in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 285–296, IEEE, 2016.

[160] Y. Yu, H. Wang, V. Filkov, P. Devanbu, and B. Vasilescu, “Wait for it: determinants of pull request evaluation latency on github,” in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pp. 367–371, IEEE, 2015.

[161] J. Tsay, L. Dabbish, and J. Herbsleb, "Influence of social and technical factors for evaluating contribution in github," in *Proceedings of the 36th international conference on Software engineering*, pp. 356–366, ACM, 2014.

[162] H. T. O. Davies, I. K. Crombie, and M. Tavakoli, "When can odds ratios mislead?," *Bmj*, vol. 316, no. 7136, pp. 989–991, 1998.

[163] D. G. Altman, J. J. Deeks, and D. L. Sackett, "Odds ratios should be avoided when events are common," *Bmj*, vol. 317, no. 7168, p. 1318, 1998.

[164] F. Tu, J. Zhu, Q. Zheng, and M. Zhou, "Be careful of when: an empirical study on time-related misuse of issue tracking data," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 307–318, ACM, 2018.

[165] S. Milborrow, "Plotting regression surfaces with plotmo," 2019.

[166] A. Liaw and M. Wiener, "Classification and regression by randomforest," *R News*, vol. 2, no. 3, pp. 18–22, 2002.

[167] H. Wickham, *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2016.

[168] J. H. Friedman, "Greedy function approximation: a gradient boosting machine," *Annals of statistics*, pp. 1189–1232, 2001.

[169] S. Frénot and J. Ponge, "LogOS: An automatic logging framework for service-oriented architectures," in *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*, pp. 224–227, IEEE, 2012.

[170] P. Boldi, A. Marino, M. Santini, and S. Vigna, "BUbiNG: Massive crawling for the masses," *ACM Transactions on the Web (TWEB)*, vol. 12, no. 2, pp. 1–26, 2018.

[171] N. Bradley, T. Fritz, and R. Holmes, "Context-aware conversational developer assistants," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 993–1003, IEEE, 2018.

[172] S. Gianvecchio, M. Xie, Z. Wu, and H. Wang, "Measurement and classification of humans and bots in Internet chat," in *USENIX security symposium*, pp. 155–170, 2008.

[173] U. Farooq and J. Grudin, “Human-computer integration,” *interactions*, vol. 23, pp. 26–32, Oct. 2016.

[174] C. Lebeuf, M.-A. Storey, and A. Zagalsky, “Software bots,” *IEEE Software*, vol. 35, no. 1, pp. 18–23, 2017.

[175] M. Monperrus, “Explainable software bot contributions: Case study of automated bug fixes,” in *2019 IEEE/ACM 1st International Workshop on Bots in Software Engineering (BotSE)*, pp. 12–15, IEEE, 2019.

[176] V. Cosentino, J. L. C. Izquierdo, and J. Cabot, “A systematic mapping study of software development with GitHub,” *IEEE Access*, vol. 5, pp. 7173–7192, 2017.

[177] T. Dey, S. Mousavi, E. Ponce, T. Fry, B. Vasilescu, A. Filippova, and A. Mockus, “tapjdey/BIMAN_bot_detection: Initial BIMAN model,” Mar. 2020.

[178] T. Dey, S. Mousavi, E. Ponce, T. Fry, B. Vasilescu, A. Filippova, and A. Mockus, “A dataset of bot commits,” Jan. 2020.

[179] T. Dey and A. Mockus, “Which pull requests get accepted and why? a study of popular npm packages,” *arXiv preprint arXiv:2003.01153*, 2020.

[180] T. Bhowmik, N. Niu, W. Wang, J.-R. C. Cheng, L. Li, and X. Cao, “Optimal group size for software change tasks: A social information foraging perspective,” *IEEE transactions on cybernetics*, vol. 46, no. 8, pp. 1784–1795, 2015.

[181] M. Zhou and A. Mockus, “Developer fluency: Achieving true mastery in software projects,” in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 137–146, ACM, 2010.

[182] A. Mockus, “Succession: Measuring transfer of code and developer productivity,” in *Proceedings of the 31st International Conference on Software Engineering*, pp. 67–77, IEEE Computer Society, 2009.

[183] N. Kerzazi and I. El Asri, “Knowledge flows within open source software projects: A social network perspective,” in *International Symposium on Ubiquitous Networking*, pp. 247–258, Springer, 2016.

[184] T. Wolf, A. Schroter, D. Damian, and T. Nguyen, “Predicting build failures using social network analysis on developer communication,” in *Proceedings of the 31st International Conference on Software Engineering*, pp. 1–11, IEEE Computer Society, 2009.

[185] M. Alan, “Turing,” *Computing machinery and intelligence. Mind*, vol. 59, no. 236, pp. 433–460, 1950.

[186] N. Winarsky, B. Mark, and H. Kressel, “The development of Siri and the SRI Venture Creation Process,” *SRI International, Menlo Park, USA, Tech. Rep*, 2012.

[187] N. Statt, “Why Google’s fancy new AI assistant is just called ‘Google’,” *Retrieved March*, vol. 21, p. 2017, 2016.

[188] A. Kerry, R. Ellis, and S. Bull, “Conversational agents in e-learning,” in *International Conference on Innovative Techniques and Applications of Artificial Intelligence*, pp. 169–182, Springer, 2008.

[189] L. Benotti, M. C. Martínez, and F. Schapachnik, “Engaging high school students using chatbots,” in *Proceedings of the 2014 conference on Innovation & technology in computer science education*, pp. 63–68, ACM, 2014.

[190] N. Thomas, “An e-business chatbot using AIML and LSA,” in *2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pp. 2740–2742, IEEE, 2016.

[191] M. Jain, R. Kota, P. Kumar, and S. N. Patel, “Convey: Exploring the use of a context view for chatbots,” in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, p. 468, ACM, 2018.

[192] N. Abokhodair, D. Yoo, and D. W. McDonald, “Dissecting a social botnet: Growth, content and influence in Twitter,” in *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*, pp. 839–851, ACM, 2015.

[193] M. Wessel, B. M. de Souza, I. Steinmacher, I. S. Wiese, I. Polato, A. P. Chaves, and M. A. Gerosa, “The power of bots: Characterizing and understanding bots in OSS projects,” *Proceedings of the ACM on Human-Computer Interaction*, vol. 2, no. CSCW, p. 182, 2018.

[194] M.-A. Storey and A. Zagalsky, “Disrupting developer productivity one bot at a time,” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 928–931, ACM, 2016.

[195] S. Pérez-Soler, E. Guerra, J. de Lara, and F. Jurado, “The rise of the (modelling) bots: Towards assisted modelling via social networks,” in *Proceedings of the 32nd*

IEEE/ACM International Conference on Automated Software Engineering, pp. 723–728, IEEE Press, 2017.

[196] I. Beschastnikh, M. F. Lungu, and Y. Zhuang, "Accelerating software engineering research adoption with analysis bots," in *2017 IEEE/ACM 39th International Conference on Software Engineering: New Ideas and Emerging Technologies Results Track (ICSE-NIER)*, pp. 35–38, IEEE, 2017.

[197] L. Erlenhov, F. G. de Oliveira Neto, R. Scandariato, and P. Leitner, "Current and future bots in software development," in *Proceedings of the 1st International Workshop on Bots in Software Engineering*, pp. 7–11, IEEE Press, 2019.

[198] C. R. Lebeuf, *A taxonomy of software bots: Towards a deeper understanding of software bot characteristics*. PhD thesis, 2018.

[199] S. Amreen, A. Mockus, R. Zaretzki, C. Bogart, and Y. Zhang, "ALFAA: Active Learning Fingerprint based Anti-Aliasing for correcting developer identity errors in version control systems," *Empirical Software Engineering*, pp. 1–32, 2019.

[200] C. Paul, A. Rettinger, A. Mogadala, C. A. Knoblock, and P. Szekely, "Efficient graph-based document similarity," in *European Semantic Web Conference*, pp. 334–349, Springer, 2016.

[201] S. Helmer, "Measuring the structural similarity of semistructured documents using entropy," in *Proceedings of the 33rd international conference on Very large data bases*, pp. 1022–1032, VLDB Endowment, 2007.

[202] D. Buttler, "A short survey of document structure similarity algorithms," tech. rep., Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2004.

[203] A. Karwath and K. Kersting, "Relational sequence alignment," *MLG 2006*, p. 149, 2006.

[204] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of molecular biology*, vol. 48, no. 3, pp. 443–453, 1970.

[205] T. F. Smith, M. S. Waterman, *et al.*, "Identification of common molecular subsequences," *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.

[206] T. Fry, T. Dey, A. Karnauch, and A. Mockus, “A dataset and an approach for identity resolution of 38 million author ids extracted from 2b git commits,” *arXiv preprint arXiv:2003.08349*, 2020.

[207] E. Guzman, D. Azócar, and Y. Li, “Sentiment analysis of commit comments in GitHub: An empirical study,” in *Proceedings of the 11th Working Conference on Mining Software Repositories*, pp. 352–355, 2014.

[208] P. G. Efthimion, S. Payne, and N. Proferes, “Supervised machine learning bot detection techniques to identify social Twitter bots,” *SMU Data Science Review*, vol. 1, no. 2, p. 5, 2018.

APPENDICES

APPENDIX A

A SYSTEMATIC METHOD FOR DETECTING BOTS THAT COMMIT CODE IN SOCIAL CODING PLATFORMS

A.1 Overview

During the previous studies, we have repeatedly observed how the presence of bots in a dataset can potentially lead to misleading conclusions. However, in order to "clean" the corresponding datasets we first need to know which of the developers in the dataset are bots, a task that is more difficult than it seems. In this chapter, we explore a novel technique for detecting bots, specifically, those who commit code in different social coding platforms. Thereby, we're trying to answer the fifth and final research topic mentioned in subsection 1.2.1 of Chapter 1: "Designing a systematic method for identifying bots that commit code to various social-coding platforms." Most of the research about this topic was published in a conference paper [34].

A.2 Introduction

Bot is a classification assigned to a software application that performs automated tasks based on a predefined set of instructions, and it either runs continuously or is triggered by events associated with events, time conditions, or manual execution. Examples of applications that can function as bots are automated scripts, activity loggers [169], web crawlers [170], and chat bots [171, 172]. A large number of software developers, teams, and companies use bots to do various, often repetitive, tasks, because bots can perform those tasks more efficiently than human users [173, 174, 175].

In social coding platforms [143, 176] such as GitHub and BitBucket a number of bots regularly create code commits, issues, and pull requests. However, detecting a bot is a challenging task because on the surface there is no apparent difference between the activity of a bot and that of a human. Moreover, the message structure, message content, and linguistic style of a code commit created by a bot can look very similar to a commit created by a human author. While there are a number of well-known and active bots, such as

Dependabot[1] and Greenkeeper,[2] not all bots are as popular and easily recognizable, as we disclose in this work.

Our review of the existing literature did not reveal any systematic approach for determining whether a given author in a social coding platform is a bot. Therefore, in this work, we propose **BIMAN** — *Bot Identification by commit Message, commit Association, and author Name* — a novel technique to detect bots that commit code. **BIMAN** is comprised of three methods that consider independent aspects of the commits made by a particular author: 1) *Commit Message:* Identify if commit messages are being generated from templates; 2) *Commit Association:* Predict if an author is a bot using a random forest model, with features related to files and projects associated with the commits as predictors; and 3) *Author Name:* Match author's name and email to common bot patterns. The code for **BIMAN** is available at our GitHub repository.[3]

We applied **BIMAN** to the *World of Code* dataset [30], which has a collection of more than 34 million authors who have committed code to a GitHub repository, along with detailed information for approximately 1.6 billion commits made by these authors. A **dataset** was compiled with information about 461 bots, detected by **BIMAN** and manually verified as bots, each with more than 1,000 commits, along with detailed information about 13,762,430 commits made by these bots. This dataset is available at DOI 10.5281/zenodo.3610205 [178].

A.3 Motivation and Research Questions

The main motivation behind our bot detection effort is twofold: 1) Data cleaning: the automated nature of bots can significantly affect the estimates of team size, the amount of activity, and developer productivity, which can threaten the validity of such measures and any decision based on such measures; and 2) Research: enabling further research into bots.

Many software researchers look at the activity of software developers for understanding their cultural behavior [32, 20, 179, 31, 26, 27], estimating team size [180], measuring productivity [181], and studying developer interaction such as knowledge flow within a project [182, 183] and prediction of build failures [184]. While conducting such studies, it is important to account for developers that are bots because bots typically have different

[1]https://dependabot.com
[2]https://greenkeeper.io
[3]https://github.com/ssc-oscar/BIMAN_bot_detection [177]

activity patterns than humans. For example, bots may generate physically impossible metrics of activity and productivity, or could at least significantly bias these estimates. Furthermore, the desire to stand out can lead to creation of extreme numbers of files or commits via automation (e.g., GitHub author *one-million-repo*[4] has 1,102,551 commits and the repository *biggest-repo-ever* [5] has 9,960,000 commits). [6]

However, the first step in adopting a data cleaning scheme to mitigate the effects of bots in software engineering research is to find the bots and, as mentioned earlier, we found no systematic approach for that. Therefore, the first research question we address in this study is: **RQ: How can we determine if a particular author is a bot?**

A.4 Related Works

The idea of "bots", or software applications that can imitate human activity, dates back to 1950 with Alan Turing asking the question "Can machines think?" [185]. Recent advancements in artificial intelligence, especially natural language processing and machine learning have led to a proliferation of bots across domains, such as in virtual assistants (Apple's Siri [186] and Google's Assistant [187], and Amazon's Alexa [7]), education [188, 189], e-commerce [190], customer service [191], and social media platforms [192].

Open-source software (OSS) communities, and software engineering in general, primarily use bots to reduce the workload of repetitive tasks. Wessel et. al. [193] studied 351 GitHub projects with more than 2,500 stars and found that 26% of them use bots, with bot usage rising since 2013. Bots also support communication and decision making [194, 195], automate deployment and evaluation of software [196], and automate tasks that would require human interaction in collaborative software development platforms [174].

However, while these studies highlight how bots are used and how prevalent bot adoption is in popular OSS projects, they do not present any generalizable method to detect the bots that are already present. Wessel et. al. [193] inspected the GitHub account of a suspected bot and checked if it is tagged as a bot. They also examined pull request messages, in search of obvious messages, for instance: "This is an automated pull request to...". Erlenhov et. al. [197] and Lebeuf [198] analyzed 11 and 3 well-known bots,

[4]https://github.com/one-million-repo
[5]https://github.com/one-million-repo/biggest-repo-ever
[6]https://bitbucket.org/swsc/overview/src/master/fun
[7]https://developer.amazon.com/en-US/alexa

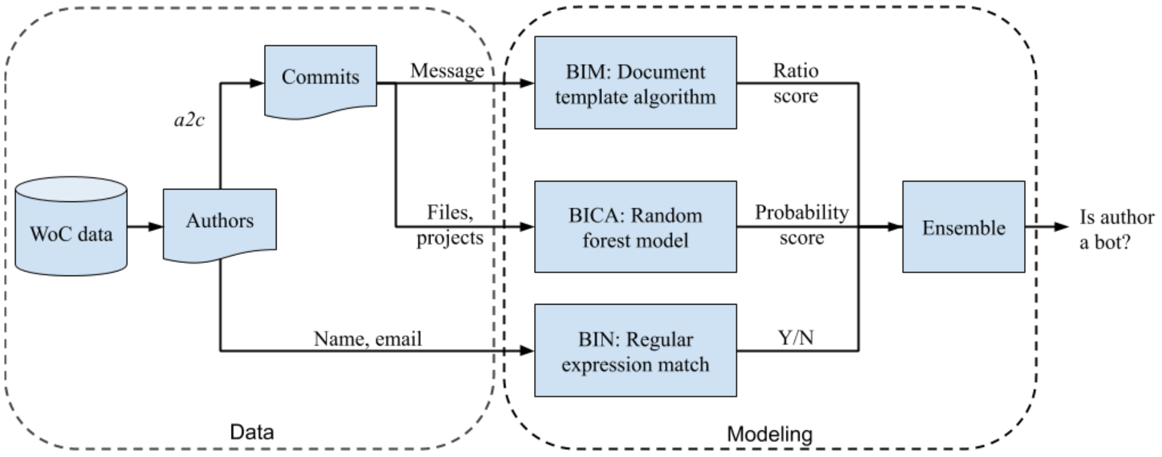


Figure A-1. BIMAN workflow: Commit data pertaining to authors is used for message template detection, activity pattern based predictions using a random forest model, and name pattern matching. Scores from each method are used by an ensemble model (another random forest model) that classifies the given author as a bot or not a bot.

respectively, and neither suggested a formal method of detecting if a given author is a bot.

A.5 Methodology

In this section, we describe the data used for analysis and present our proposed approach for detecting bots.

A.5.1 Data

The data used for this study was obtained from the World of Code (WoC) [30] dataset. Specifically, version P which was collected between May 15, 2019 and June 5, 2019 based on updates/new repositories identified on May 15, 2019. The data contained information on 73,314,320 unique non-forked Git repositories, 34,424,362 unique author IDs, and 1,685,985,529 commits. The author IDs were represented by a combination of the authors name and email address: `first-name last-name<email-address>`. As an example, for an author with first name “John”, last name “Doe”, and email address “myemail@me.com”, the corresponding author ID in the WoC dataset would be: “`John Doe<myemail@me.com>`”.

The data is stored in the form of mappings between various Git objects. For our study we used the mappings between the commit authors and commits (*a2c*), commits and filenames (including the file path) changed by that commit (*c2f*), commits and the GitHub projects that commit is associated with (*c2p*), and commits and the contents of the commits comprising the commit timestamp, timezone, and commit message (*c2cc*).

Our method of extracting information about the authors consisted of the following steps:

1. Obtaining a list of all authors from the WoC dataset.
2. Identifying all commits for the authors using the *a2c* map.
3. Extracting the list of files modified by a commit, the list of projects the commit is associated with, and the commit content for each of the commits for every author using the *c2f, c2p*, and *c2cc* maps, respectively.

A.5.2 Bot Detection

BIMAN, our proposed technique for detecting bots, comprises three methods, 1) Bot Identification by Name (**BIN**), 2) Bot Identification by commit Message (**BIM**), and 3) Bot Identification by Commit Association (**BICA**), each relying on distinct data attributes. We discuss these methods separately rather than as a single model because they can be used independently and not all of the required data for each method is available or easily obtainable, and researchers with access to partial data can still use a subset of **BIMAN**. An overview of the **BIMAN** approach is illustrated in Figure A-1.

*A.5.2.1 Identifying bots by name (**BIN**)*

We began devising a possible method for detecting bots by comparing names of known bots. Erlenhov et. al. [197] studied 11 bots, which we took as a starting point in our investigation. However, since the author IDs in our dataset consist of name-email combinations, we had to search through the list of authors for identifying the possible author IDs that could be related to one of these 11 bots. We did not found an entry matching 3 of the 11 bots: “First-timers”, “Marbot”, and “CssRooster”, and found a total of 57 author IDs that could be associated with one of the other bots. We noticed that 25 (37%) of these author IDs had the substring “bot”. We further searched for other known bots (e.g., Travis CI and Jenkins bots) in our dataset and noticed that many of the author IDs we suspected as bots also had the substring “bot” in their name or email.

Based on these observations, regular expressions were used to identify if an author is a bot by checking if the author name or email has the substring “bot”. However, to avoid including false positives like “Abbot” or “Botha”, the regular expression searched for “bot” preceded and followed by non-alpha characters. We further excluded author IDs that had the word “bot” only in the domain name of their email addresses (e.g., hr@future-*bot*.ai), since we are not convinced that these are always bots. Although matching regular expressions does not detect all bots, nor is it able to filter authors trying to disguise themselves as bots, it is a straightforward solution that does not requires any other data, and can be regarded as a good starting point.

Creating the Golden Dataset for BIM and BICA: There is no publicly available *golden dataset* of bots in social-coding platforms for training machine learning models. However, we noticed that the name based bot identification method was very precise, i.e., it had few false positives. Therefore, we used **BIN** to create a *golden dataset*. Two of this paper’s authors independently analyzed the author IDs and descriptions, and commit and pull request messages, when available, to manually verify the authors identified as bots by **BIN** and remove the ambiguous cases (less than 1%) based on consensus. We found a total of 13,150 bot authors via this process. We also needed to include a set of human authors to complete a training dataset. We randomly selected 13,150 authors, and again manually ensured that no bots were in this list. This was our *golden dataset*, consisting of 26,300 authors, used for training and testing the **BIM** and **BICA** methods of **BIMAN**.

Comparing the commit activities of humans and bots: Our initial assumption was that bots are very active agents and produce a significantly greater number of commits than humans, therefore, we could detect bots by evaluating the number of commits. However, upon investigating the 13,150 bots in the *golden dataset*, we found that assumption to be incorrect. While the maximum number of bot commits was admittedly huge ($2,463,758$), the median number was only 2, and the first and third quantile values were 1 and 16, respectively. In contrast, the median number of human commits was 4, and the first and third quantile values were 2 and 17, respectively. These observations indicated that the number of commits between humans and bots is not significantly different.

We hypothesize that the reason behind why many bots have few commits relates to any of the following reasons: (1) Given that author IDs consist of a name-email combination, slight variations in either appear as different authors, when they are not. For example, a “dotnet-bot” has three name variations that appear as different authors: `beep boop`, `Beep boop`, and `Beep Boop`, though it has the same email address: `dotnet-bot@microsoft.com`. We need to employ anti-aliasing methods [199] to address this issue. (2) Bots might have been implemented as an experiment or coursework, and never used afterwards. For example, we found a bot named "learn.chat.bot" that most likely belongs in this category. (3) Bots might have been designed for a project, but were never fully adopted.

*A.5.2.2 Detecting bots by commit messages (**BIM**)*

Characteristics of commit messages can be used to identify an author as a bot. One approach is to assume that many bots routinely use template messages as the starting point for the commit message. Consequently, detecting if the commit messages by an author originate from a template can be used to identify such bots. Although humans can also generate commit messages with similar and consistent patterns (e.g., follow a set of software development guidelines), the key assumption **BIM** follows is that for a large number of commit messages, the variability of messages' content generated by bots is lower than those generated by humans.

BIM utilizes the document template score algorithm presented in Alg. 1. Given a set of documents (commit messages), the algorithm compares document pairs and uses a similarity measure to group documents. The `Similarity` procedure represents a method that computes a "similarity" measure that is of interest [200, 201, 202, 203], with the percent identity of the aligned commit messages being used for **BIM**. A group represents documents that are suspected to conform to a similar base document, and each group has a single *template document* assigned to it and this is the document always used for comparisons. A new group is created when a document's similarity with any *template document* does not reach the similarity threshold, k_b, and this document is set as the *template document* for that group. After all documents are compared, a score is calculated based on the ratio of the number of *template documents* and the number of documents: $1 - \frac{\|T\|}{\|D\|}$, where T is the set of *template documents* and D is the set of documents.

In **BIM**, commit messages were aligned and scored using a combination of global (Needleman-Wunsch [204]) and local (Smith-Waterman [205]) sequence alignment algorithms available via the Python `alignment`[8] library. The similarity threshold, k_b, was set to 40% after testing the accuracy of Alg. 1 on the golden data using thresholds of 40, 50, 60, and 70%.

[8]https://pypi.org/project/alignment

Algorithm 1 Document template score

```
    Inputs: set of documents $D$ and similarity threshold $k_b$
    Output: 1 - ratio of number of templates to documents
 1: $T \leftarrow \emptyset$                                  ▷ set of template documents
 2: $\boldsymbol{G} \leftarrow \{\emptyset\}$      ▷ template groups, $\boldsymbol{G}_i$ is associated to template $i$
 3: for $d \in D$ do
 4:     for $t \in T$ and $d \notin \boldsymbol{G}$ do
 5:         if SIMILARITY($d$, $t$) $> k_b$ then
 6:             Add $d$ to $\boldsymbol{G}_t$
 7:         end if
 8:     end for
 9:     if $d \notin \boldsymbol{G}$ then
10:         Add $d$ to $T$
11:         Add $d$ to $\boldsymbol{G}_d$
12:     end if
13: end for
14: return $1 - \frac{\|T\|}{\|D\|}$
```

*A.5.2.3 Detecting bots by files changed and projects associated with commits (**BICA**)*

We calculated 20 metrics using the files changed by each commit, the projects that commit is associated with, and the timestamp and timezone of the commits, based on our initial assumptions about how bots and humans might be different, and empirical validation of the assumption by observing the differences in distribution of those variables for bots and humans.

For predicting whether an author is a bot using the numerical features, we tested several modeling approaches: linear and logistic regression, generalized additive models, support vector machines, and random forest. The random forest model performed better than the other approaches, so we decided to use that approach. We used the random forest implementation available in the “randomForest” package in R, with these 20 variables as predictors, to predict if the author of those commits is a bot. After iteratively selecting and removing predictors based on their importance in the model, and measuring the AUC-ROC every time, we found that a model with only 6 predictors was the best model. The list of predictors is given in Table A-1, along with the description of each variable. We found that the timestamp of a commit and any time related measure (e.g., how long a bot has been active and at what times of the day it makes commits) are not important predictors.

In order to tune the random forest model, we used the "train" function from the *caret* package in R for performing a grid search (using a 10 fold cross-validation) on the training data to find the best values of the model parameters that resulted with the highest accuracy: "ntree" (number of trees to grow) and "mtry" (number of variables randomly sampled as candidates at each split). The optimum values for "ntree" and "mtry" were 100 and 2, respectively.

Table A-1. Predictors used in the random forest model

Variable Name	Variable Description
Tot.FilesChanged	Number of files changed by author across commits (includes duplicates)
Uniq.File.Exten	Number of unique file extensions in all the author's commits
Std.File.pCommit	Std. dev. of number of files per commit
Avg.File.pCommit	Mean number of files per commit
Tot.uniq.Projects	Number of unique projects commits have been associated with
Median.Project. pCommit	Median number of projects the commits have been associated with (includes duplicates); We took the median value, because the distribution of projects per commit was very skewed, and the mean was heavily influenced by the maximum value.

A.5.2.4 Ensemble model:

Based on the fact that **BIN**, **BIM**, and **BICA** methods consider different aspects of the authors and commits, we decided to use an ensemble model, implemented as another random forest model. The ensemble model in **BIMAN** utilizes the outputs of the three methods as predictors to make a final judgement as to whether an author is a bot or not. The output from **BIN** is a binary value ($1 \rightarrow$ bot, $0 \rightarrow$ human), stating if the author ID matches the regular expressions we checked against; the output from **BIM** is a score, with higher values corresponding to a higher probability of the author being a bot; and the output from **BICA** is the probability that an author is a bot.

Creating the Training Dataset for the ensemble model: Recall the *golden dataset* was generated using the **BIN** method, so we did not used it for training the ensemble model. Instead, we created a new training dataset partly consisting of 67 bots from which 57 author IDs were associated with 8 bots described in [197] (as mentioned in Section A.5.2.1) and 10 author IDs associated with 3 other known bots that were not in the *golden dataset*: *Scala Steward*, *codacy-badger*, and *fossabot*. Also, 67 human authors were included via random selection and manual validation. The final training data for the ensemble model had only

134 observations, however, given that we had 3 predictors, we were reasonably satisfied with it.

Table A-2. Performance of the models in detecting 8 known bots from [197] and 3 other known bots outside the *golden dataset*

Bot	**No. of author IDs associated with the bot**	**No. of IDs identified as bot by BIN**	**No. of IDs identified as bot by BICA**	**No. of IDs identified as bot by BIM**	**No. of IDs identified as bot by BIMAN**
Dependabot	4	4	4	2	4
Greenkeeper	15	10	13	11	13
Spotbot	1	0	1	1	1
Imgbot	5	1	4	3	4
Deploybot	29	9	20	17	23
Repairnator	1	0	0	1	1
Mary-Poppins	1	0	1	1	1
Typot	1	1	0	1	1
Scala Steward	6	0	6	6	6
codacy-badger	2	0	2	2	2
fossabot	2	0	2	2	2
Total	**67**	**25 (37%)**	**53 (79%)**	**47 (70%)**	**58 (87%)**

A.6 Results

A.6.1 Qualitative Validation of BIMAN

Before going into the detailed performance evaluation of **BIMAN**, we wanted to test how it performs in detecting a few known bots. As mentioned in Section A.5.2.1, we obtained a set of 57 author IDs associated with 8 of the bots described in [197]. In addition, we examined 10 author IDs associated with 3 well-known bots, *Scala Steward*, *codacy-badger*, and *fossabot*, that were not in the *golden dataset*. The performance of bot detection of **BIMAN** and each of its constituent methods is shown in Table A-2.

We found that **BIMAN** identified 58 (87%) out of 67 author IDs as bots, and 6 out of 9 other IDs could be identified as not actually being a bot via manual investigation, they were either spoofing the name or simply using the same name. The 3 other IDs, 2 of which were associated with "Deploybot" [9] [10], and the other with "Imgbot" [11], had 1 commit each, making any decision about them being bots difficult to make even via manual investigation.

[9]deploybot-lm <45803032+deploybot-lm@users.noreply.github.com>

[10]DeployBot <deploybot@imqs.co.za>

[11]imgbot<imgbothelp@gmail.com>

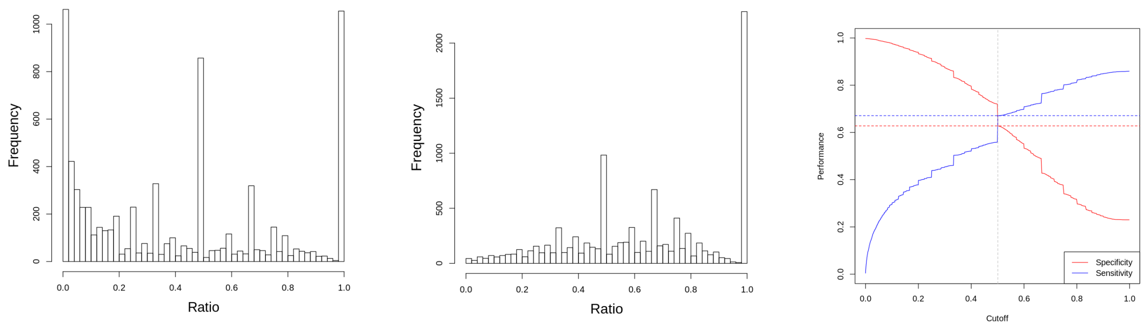


Figure A-2. (a) Ratio of number of detected templates and the number of commit messages for the 13,150 bots in the *golden dataset*; (b) Ratio of number of detected templates and the number of commit messages for the 13,150 humans (non-bots) in the *golden dataset*; (c) Plot of sensitivity, specificity, and cutoff (threshold), when predicting if an author is a bot using the ratio of number of detected templates to the number of commit messages.

A.6.2 Performance Evaluation of BIMAN

In this section, we discuss the performance of **BIMAN**, our proposed approach for bot detection. As mentioned in Section A.5.2, **BIMAN** is comprised of three independent methods, each looking at a different aspect of the commit authors and the commits, and an ensemble model that combines the results from the three methods for estimating the final prediction. We decided to evaluate the performance of each method and discuss what we learned with each one in detecting bots that commit code.

*A.6.2.1 Performance of **BIN**:*

We did not use the *golden dataset* to validate the accuracy of **BIN** because this method was used to construct that dataset (see Section A.5.2.1). However, during creation of the *golden dataset*, **BIN** obtained a precision close to 99%, which indicates that any author considered to be a bot using this method has a very high probability of being a bot. In general, humans do not try to disguise themselves as bots. The recall measure is not high, because **BIN** missed a lot of cases where the bots do not explicitly have the substring “bot” in their name. As mentioned in Section A.6.1, we observed an estimated recall of 37% on the set of 67 bot IDs we manually investigated.

*A.6.2.2 Performance of **BIM**:*

Our proposed method for detecting whether an author is a bot by applying the document template score algorithm, Alg. 1, solely relied on the commit messages. Figures A-2a-b present the ratio of the number of probable templates to the number of commit messages for the bots and humans in the *golden dataset*. Note that bots tend to have a lower ratio than humans. The reason for both plots having a high template ratio is that if an author has a single commit message, the ratio is bound to be 1. Over 25% of the bots in the *golden dataset* have only 1 commit.

We wanted to find an optimal threshold for the template ratio, so that the authors, for whom the ratio is lower than the threshold, can be regarded as bots (the output from the **BIM** method is $(1-\text{ratio})$, so a lower value is more likely to be human and vice versa). This information would be useful for researchers who might only use this technique to detect whether a given author is a bot, and this also helps us calculate the performance of this method. The optimal threshold was found using the "closest.topleft" optimality criterion (calculated as: $min((1-sensitivities)^2+(1-specificities)^2)$) using the *pROC* package in R. The AUC-ROC value using the ratio values as predicted probabilities was 0.70, and the optimal values for the threshold, sensitivity, and specificity were found to be 0.51, 0.67, and 0.63, respectively. We plotted the sensitivity and the specificity measures in Figure A-2c, and highlighted the optimal threshold, sensitivity, and specificity values for that threshold.

True Positive: The cases where this model could correctly identify bots were cases where the bots actually used templates or repeated the same commit message, e.g., a bot named "Autobuild bot on Travis-CI" used the same commit message "`update html,`" for all of the 739 commits it made, and a bot named "Common Workflow Language project bot" created 1,373 commits that used the form: "`$USER-CODE: $SOFTWARE configuration files updated. Change performed by $NAME`". **BIM** could identify these messages as coming from the same template message and classify these authors as bots.

False Negative: The cases where this model could not correctly identify bots were mostly cases where the bots reviewed code added by humans and created a commit message that added a few words with the commit message written by a human, e.g., a bot named "Auto Roll Bot" created commit messages in the form of: "`$COMMIT-SEQUENCE-NUMBER: $LONG-HUMAN-COMMIT-TEXT $PATTERN`", with one specific example being "`3602: Fix errors in the Newspeak Mac installer genrators.`

`Fix a slip in platforms/Cross/vm/ sqCogStackAlignment.h for the ARM's getsp.  Eliminate non-spur and stack VMs from the ARM builds (it builds veerry slowly) Include 64-bit and Mac Pharo VMs in archives and uploads.---------------------`", with the length of "$LONG-HUMAN-COMMIT-TEXT" typically ranging between 20 and 50 words. **BIM** failed to identify this template and misclassified this author as a human.

True Negative: The human authors correctly identified had some variation in the text, with the usual descriptions of change. Some examples are: "`Added a count down controller`" and "`Enabling multiport deployments.  By mapping ports a little bit more specific we get all the servers listed in the server browser`".

False Positive: In contrast, humans who were misclassified as bots usually had short commit messages that were not descriptive, and they reused the same commit message multiple times. Example of typical messages are: "`Initial Commit`", "`Added File by Upload`", and "`Updated $FILE`".

Our observations support that **BIM** is useful in detecting "typical" bots that modify small parts of a message in every commit, and "typical" humans who write descriptive commit messages. However, we can also conclude that it is very hard to identify if an author is a bot using just one signal.

A.6.2.3 Performance of ***BICA****:*

As mentioned in Section A.5.2.3, the **BICA** approach uses a random forest model with the measures listed in Table A-1 as predictors for identifying bots. We used the *golden dataset* generated using the **BIN** method (Section A.5.2.1) for training the model and testing its performance. 70% of the data, selected randomly, was used for training the model and the rest 30% was used for testing it, and the procedure was repeated 100 times with different random seeds.

The model showed good performance, with an AUC-ROC value of 0.89. The variable importance plot (Figure A-3) indicates that the total number of unique file extensions and the total number of files changed in all the commits made by an author are the most important variables.

To understand what each of the predictors tell us about how the behavior of the bots differ from that of humans, we looked at their partial dependence plots, see Figure A-4.

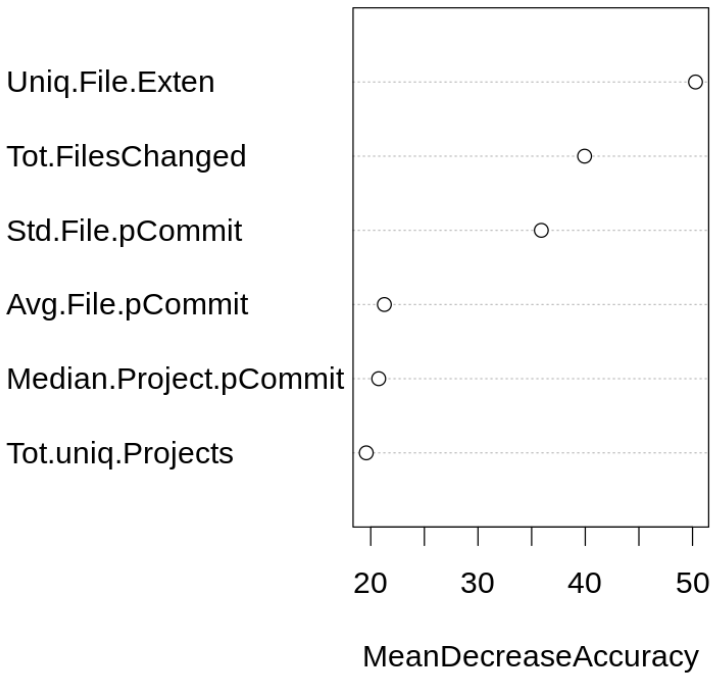


Figure A-3. Variable importance plot for the random forest model used to identify bots

The greater values in the vertical axis of each plot correspond to a higher probability of an author being a bot, and the values in the horizontal axis are the possible predictors' values. These plots illustrate an empirical understanding about how the behavior of the bots is different from humans. We notice that bots tend to have fewer number of unique file extensions and their commits are associated with fewer number of different projects, i.e., they tend to operate in one ecosystem. However, their commits tend to be associated with a greater number of projects per commit, the projects they commit to are more popular. Bots typically make larger commits, as we notice that they tend to have more files per commit on average and a greater number of total files changed. They are also more consistent in terms of commit size because the variation in the number of files per commit is lower. These observations fall in line with our idea of typical bots, which keep updating a consistent set of files and typically partake in popular projects.

A.6.2.4 Performance of the ensemble model:

We combined the results of **BIN**, **BIM**, and **BICA** using an ensemble model, implemented as a random forest model. The dataset used for training and testing the performance of this model had only 134 observations, because of reasons described in

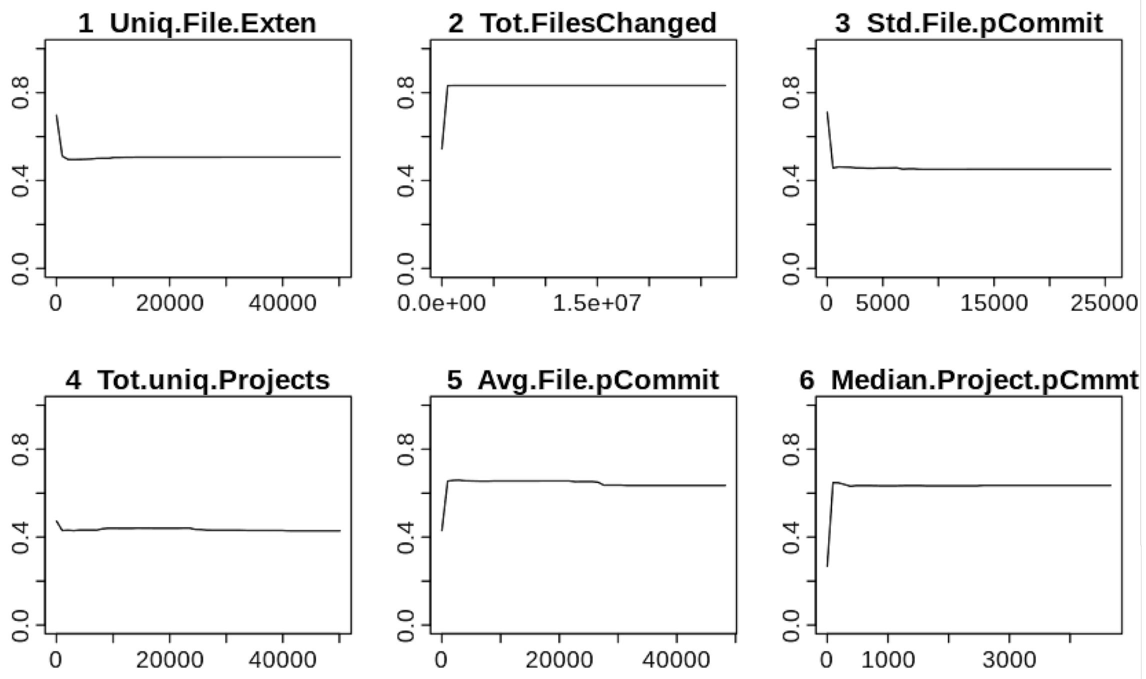


Figure A-4. Partial dependence plot for all the predictors in the random forest model

Section A.5.2.4. We used 80% of the data for training, and 20% for testing, and repeated the process 100 times with different random seeds. The performance of this model had variation because of the small size of the training data. The value of the AUC-ROC measure varied between 0.89 and 0.95, with a median of 0.90.

*To address the RQ, we devised **BIMAN**, a systematic approach for detecting bots using information about their names, commit messages, files modified by the commit, and projects associated with the commits.*

A.6.3 Estimating the Number of Commit Bots

While we can easily obtain the number and activity of author IDs that contain the substring "bot", it is much more difficult to determine the total number of author IDs that, from their string representation, can not be inferred to be bots. Yet, even a rough gauge on the prevalence of bots and code committed by bots would be helpful to have a handle on the fraction of code commit activity that is automated. To do that we randomly selected a sample of 10,000 authors IDs outside of our datasets used so far (none had the substring "bot" in their names). **BIMAN** predicted 1,167 authors IDs to be bots, and we randomly sampled 100 authors IDs among those. A subjective assessment conducted by two authors of this paper discovered at least 9 of these author IDs likely to produce mostly automated commits. From this, we can obtain a rough estimate that approximately $11.67\% \times 9\% \approx 1\%$ of all authors IDs who commit code are bots. Therefore, from the total population of approximately 40 million authors in open-source Git commits, approximately 400,000 authors are bots. The 9 author IDs that we identified as bots were found to have created between 10 and 1,500 times more commits than the remaining author IDs. Such high discrepancy strengthens our concerns described in Section A.3 that the empirical analyses relying on measures of developer productivity can be strongly biased even if the actual bot population represents a modest 1% of all developer author IDs.

A.6.4 Shared Dataset of Bot Commits

We have compiled the information about the commits made by 461 bots detected using **BIMAN**, each of whom have created more than 1,000 commits, into a single dataset and made it available for researchers interested in conducting studies on such data, which includes information about $13,762,430$ commits made by these bots. We decided to focus on the more active bots since these bots would have a much greater effect on any estimate of developer productivity, team size, etc. and they are the ones that should be accounted for during any data cleaning process.

The data is stored in a delimited text file (semicolon as the separator) with the following format in each line: "author_id"; "commit-sha"; "time-of-the-commit"; "timezone"; "files-modified-by-the-commit"; "projects-the-commit-is-associated-with"; "commit-message". In the case of having multiple projects and files for a given commit, they are separated by ','. The data is available at *Zenodo*, through the link provided in Section A.2. Additional data about other authors, along with the likelihood of each being a bot will be provided upon requests.

A.7 Limitations and Future Work

Our approach of detecting bots is a first step towards a challenging task, and there are a number of limitations to our approach and possible scope for improvement.

A.7.1 Internal Validity

The biggest problem we faced during designing **BIMAN** was the lack of a golden dataset. We only knew about a handful of bots, which was not enough to design an accurate machine learning model. We tackled this problem by creating a dataset with one of the methods we proposed (**BIN**), and manually validating it. However, the bots found by **BIN** can have different characteristics than the rest of the bots, specifically, ones that may be trying to hide the fact that they are bots, and our method is not be able to detect them.

We did not have a ground truth to validate the *golden data* against (nor are we aware of the possibility of compiling such data with absolute certainty), so we had to come up with, what we judged to be, a reasonable alternative. The *golden dataset* was manually curated by two authors of this paper, and the ambiguous cases, including the bots trying to disguise themselves as humans and vice versa, were excluded from the *golden dataset*.

Using the dataset generated by **BIN** as a golden dataset also means that we were not able to estimate the recall of **BIN** with it. Instead, we had to use a much smaller dataset to estimate its recall. Similarly, our final ensemble model was also trained with this smaller dataset, which led to some variation in its performance (AUC-ROC value varied between 0.89 and 0.95).

Another threat to the effectiveness of our method is that a number of developers use automated scripts to handle some of their works, which uses their Git credentials while making commits. This is a tricky challenge for our method, since the signal from those authors appears mixed, and depending on what fraction of commits made using that author's ID is made by the bots, our method can fail to detect such authors as bots. Similarly, a few organizational IDs are sometimes used by bots as well as humans, and we have a similar issue regarding those IDs as well. We did not address the problem of multiple IDs belonging to the same author, however, we are testing different approaches for addressing this issue [206], and, as a future work, plan on extending **BIMAN** with this capability.

Provided that an estimated 1% of the commit authors were found to be bots (Section A.6.3), an author detected as a bot by a 90% accurate model has only only 8.3% chance of actually being a bot (using Bayes' Theorem), i.e., we are bound to have false positives.

A.7.2 Construct Validity

The construct validity threats primarily apply to the **BIM** approach we used, since it was designed with specific ideas about how a bot might work. **BIM** focuses on identifying bots that authored all the commit messages it is associated with, independent of whether they were generated by a template-based approach or not. However, many developers make use of bots for generating certain commit messages (re-using the same author ID) and this hybrid classification is not addressed in this work. The main factors that give rise to limitations are the content of commit messages, number of commit messages per author, and performance of similarity measure.

The performance of **BIM** depends primarily on the performance of the similarity measure used to compare commit messages. Commit messages tend to be concise, making it difficult to extract content characteristics (structural or semantic) that are useful for text similarity metrics. Humans with consistent message styles become difficult to differentiate from template-based bot messages. Moreover, if there are few commit messages or many unique messages, the document template score will not be effective, thus, this approach works better when enough data is available to almost saturate the document template score. We note that **BIM**'s performance will also vary based on the language of the commit messages (e.g., Spanish and Chinese), and does not support multilingual sets of commit messages. **BIM**'s performance can be improved by using more effective similarity measures based on natural language processing [207], document embeddings, clustering, and machine learning models [208].

A.7.3 External Validity

Our goal in this paper was to identify bots that make commits in social coding platforms. To that effect, our method of detecting bots could work for detecting other types of bots, such as pull-request bots, and chat bots.

A.8 Summary: Addressing RT-5

The goal of this study was to address the fifth research topic mentioned in subsection 1.2.1: *"Designing a systematic method for identifying bots that commit code to various social-coding platforms."*

To address this topic, we have designed a systematic method for detecting code-commit bots called **BIMAN**, which gave an AUC-ROC value of 0.94 while trying to find bots. This process can be further improved by addressing the problem of developers/ bots using multiple IDs. Using this method to find and filter out bots would greatly enhance the accuracy of empirical software engineering tasks similar to what was performed by my other projects.

APPENDIX B
LIST OF PUBLICATIONS

List of Publications related to this Research:

B.1 Book Chapters

1. Sadika Amreen, Bogdan Bichescu, Randy Bradley, **Tapajit Dey**, Yuxing Ma, Audris Mockus, Sara Mousavi, and Russell Zaretzki. "A Methodology for Measuring FLOSS Ecosystems." In "Towards Engineering Free/Libre Open Source Software (FLOSS) Ecosystems for Impact and Sustainability", pp. 1-29. Springer, Singapore, 2019. LINK

B.2 Journal Articles

1. **Tapajit Dey** and Audris Mockus. "Deriving a Usage-Independent Software Quality Metric." Empirical Software Engineering (2020): 1-46. LINK. Impact Factor: 4.457 (2018)

2. Andrey Krutauz, **Tapajit Dey**, Peter Rigby, and Audris Mockus. "Do Code Review Measures Explain the Incidence of Post-Release Defects?", Accepted in the Empirical Software Engineering journal, In Press. Impact Factor: 4.457 (2018)

B.3 Conference Articles

1. **Tapajit Dey** and Audris Mockus. "Impact of Technical and Social Factors on Pull Request Quality for the NPM Ecosystem", Accepted in The ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2020. LINK

2. **Tapajit Dey**, Sara Mousavi, Eduardo Ponce, Tanner Fry, Bogdan Vasilescu, Anna Filippova, and Audris Mockus. "Detecting and Characterizing Bots that Commit Code", Accepted in Mining Software Repositories (MSR) Conference, 2020. LINK

3. Tanner Fry, **Tapajit Dey**, Andrey Karnauch, and Audris Mockus. "A Dataset and an Approach for Identity Resolution of 38 Million Author IDs extracted from 2B

Git Commits”, Accepted in Mining Software Repositories (MSR) Conference (Data Showcase Track), 2020. LINK

4. **Tapajit Dey**, Yuxing Ma, and Audris Mockus. “Patterns of effort contribution and demand and user classification based on participation patterns in npm ecosystem." In Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering, pp. 36-45. ACM, 2019. LINK

5. **Tapajit Dey** and Audris Mockus. “Modeling Relationship between Post-Release Faults and Usage in Mobile Software." In Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering, pp. 56-65. ACM, 2018. LINK

6. **Tapajit Dey** and Audris Mockus. “Are software dependency supply chain metrics useful in predicting change of popularity of npm packages?." In Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering, pp. 66-69. ACM, 2018. LINK

7. **Tapajit Dey**, Jacob Logan Massengill, and Audris Mockus. “Analysis of popularity of game mods: A case study." In Proceedings of the 2016 Annual Symposium on Computer-Human Interaction in Play Companion Extended Abstracts, pp. 133-139. ACM, 2016. LINK

B.4 Workshop Articles

1. **Tapajit Dey**,Bogdan Vasilescu, and Audris Mockus “An Exploratory Study of Bot Commits” — Invited article in the 2nd International Workshop on Bots in Software Engineering (BotSE), 2020. LINK

2. Yuxing Ma, **Tapajit Dey**, and Audris Mockus. “Modularizing global variable in climate simulation software: position paper." In Proceedings of the International Workshop on Software Engineering for Science, pp. 8-11. ACM, 2016. LINK

B.5 Pre-print Articles (Not Peer-reviewed)

1. **Tapajit Dey** and Audris Mockus. “A Matching Based Theoretical Framework for Estimating Probability of Causation." arXiv preprint arXiv:1808.04139 (2018). — Proposing a novel way to estimate probability of causation using statistical matching for experimental and observational scenarios. LINK

2. Yuxing Ma, **Tapajit Dey**, Jarred M. Smith, Nathan Wilder, and Audris Mockus. “Crowdsourcing the discovery of software repositories in an educational environment." PeerJ Preprints 4 (2016): e2551v1. LINK

B.6 Works Under Review/In Progress

1. Yuxing Ma, **Tapajit Dey**, Chris Bogart, Sadika Amreen, Marat Valiev, Adam Tutko,d David Kennard, Russell Zaretzki, Audris Mockus. “World of Code: Enabling a Research Workflow for Mining and Analyzing the Universe of Open Source VCS data”, Under Review in the International Journal of Empirical Software Engineering journal. — A broad description of the World of Code dataset and the associated tool.

2. **Tapajit Dey**, Andrey Karnauch, and Audris Mockus. “Skill Spaces: Representation of Expertise in Open Source Software” Under review — Representing Software Developers’ and projects’ specific expertise using LSI and Doc2Vec embedding over a 200 dimensional skill space

3. **Tapajit Dey** and Audris Mockus. “Effect of Software Dependencies on Software Popularity”, In progress — A study highlighting the effect of the number of dependents and their popularity on the popularity of a software package.

VITA

Tapajit Dey grew up in a small township near the Kolaghat Thermal Power Plant in the state of West Bengal in India. After completing his schooling from a local school, he went to the Indian Institute of Technology (IIT), Kharagpur for his undergraduate studies, from where he completed his Bachelors and Masters Degree (under the 5 Year Dual Degree program) in Electronics and Electrical Communications Engineering in 2012. Then he moved to the city of Bangalore, India where he worked in the IBM India Software Labs as a Staff Research Engineer for 3 years. He joined the University of Tennessee, Knoxville in August, 2015 for his doctoral studies in Computer Science with Dr. Audris Mockus. His research interests include empirical software engineering, mining software repositories, and data analytics.