STUDYING REOPENED BUGS IN OPEN SOURCE SOFTWARE SYSTEMS

by

ANKUR TAGRA

A thesis submitted to the

School of Computing

in conformity with the requirements for

the degree of Master of Science

Queen's University

Kingston, Ontario, Canada

October 2021

Copyright © Ankur Tagra, 2021

Abstract

REOPENED bugs can degrade the overall reputation of a software system since such bugs lead to a loss of end-users trust regarding the quality of the software. Thus, understanding the characteristics of bugs that are reopened, and what factors are more likely to affect a reopened bug (especially post-release reopened bug) to be fixed rapidly throughout the release lifecycle, could provide insights in helping software developers to avoid/minimize such bugs.

In this thesis, we study the characteristics of reopened bugs and the factors that lead to a post-release reopened bug being fixed rapidly/slowly. To understand the characteristics of reopened bugs, prior studies built statistical or machine learning models to analyze the factors that impact the likelihood of a bug getting reopened. However, we observe several aspects that require further investigation from prior studies: 1) previously studied datasets are too small (only consisting of 3 projects) 2) 1 out of the 3 studied projects has a data leak issue. 3) the previously used experimental steps are

outdated. After considering such aspects, we observe that only 34% of the studied projects give an acceptable performance with AUC $\geqslant$ 0.7 for predicting if a bug will be reopened. Moreover, we observe that post-release reopened bugs take only 189.1 hours rework time (time taken to resolve a reopened bug) as compared to 388.4 hours for rework time in pre-release reopened bugs. To study the likelihood of a post-release reopened bug getting fixed rapidly, we build prediction pipelines and observe that the models give an acceptable AUC of 0.78 to determine if a post-release reopened bug will get resolved rapidly/slowly. Our model predicts if a post-release reopened bug will get resolved rapidly (i.e., less than 3 minutes) or slowly (i.e., more than 4,538 hours) by considering top 20% fast resolved bugs as rapidly resolved and bottom 20% fast resolved bugs as slowly resolved.

Based on our findings, we encourage future research to leverage the rich data available during and after a bug is reopened, to understand the eventual resolution of a reopened bug and we also encourage researchers to consider pre-release and post-release reopened bugs separately in their analysis as studying reopened bugs as a whole leads to biased implications.

Acknowledgments

First of all, I would like to applaud my collaborators Dr. Haoxiang Zhang, and Dr. Gopi Krishnan Rajbahadur for guiding me in my academic journey and providing me continuous feedback to improve. Dr. Haoxiang Zhang helped me by providing valuable feedback and helped me learn how to do research. Dr. Gopi Krishnan Rajbahadur helped me by suggesting interesting and innovative angles to my research. I learned not just technical skills but time management and presentation skills from them which helped me become a better person.

I would like to express my sincerest gratitude to my supervisor Professor Ahmed E. Hassan for his continuous motivation and support. Without his guidance, this thesis would not have been possible. Ahmed bolstered me to pursue my interests. I feel delighted to have gotten a chance to work under his supervision and learn a lot from him which I can apply in my life.

I would also like to thank my examination committee members for providing their

valuable comments and feedback.

I would like to thank Professor Ying Zou, Professor Bram Adams, and Professor Foutse Khomh for their guidance and support in my research. I am extremely fortunate to work in Software Analysis and Intelligence Lab (SAIL). I would like to thank post-doctoral researcher Dr. Gustavo A. Oliva for valuable discussions about my research. I would like to thank Dr. Mohammed Sayagh for providing support and guidance for my research. I had a great experience working with him. I would also like to thank Dr. Heng Li, Dr. Muhammad Asaduzzaman and PhDs (Kundi Yao, Jiahuei Lin, Md Ahasanuzzaman, Aaditya Bhatia, Yingzhe Lyu) and M.Sc. (Michael Pacheco) for providing a fun and amazing environment in the lab.

I would like to dedicate this thesis to my father Mr. Raj Kumar Tagra. He always backs me thus providing a stress-free environment for me to achieve my goals. I would like to thank my mother Sunita Tagra and sister Himani Tagra for believing in me and making my M.Sc. journey remarkable.

Co-authorship

In all the thesis chapters and related publications, my contributions are drafting the research plan; conducting a background study and related work; data collection; conducting the experiments and data analysis; and writing and polishing drafts.

Earlier versions of the work in the thesis are published/under review as listed below:

1. Revisiting reopened bugs in open source software systems.
   Ankur Tagra, Haoxiang Zhang, Gopi Krishnan Rajbahadur, Ahmed E. Hassan, under review for the Journal of Empirical Software Engineering.

2. Studying post release reopened bugs in open source software systems.
   Ankur Tagra, Haoxiang Zhang, Gopi Krishnan Rajbahadur, Ahmed E. Hassan, Ying Zou, to be submitted for the Journal of Empirical Software Engineering.

Table of Contents

Abstract i

Acknowledgments iii

Co-authorship v

List of Tables viii

List of Figures ix

1 Introduction 1
- 1.1 Thesis Statement . . . 3
- 1.2 Thesis Overview . . . 3
- 1.3 Thesis Contributions . . . 6

2 Background and Related Work 7
- 2.1 Background . . . 7
- 2.2 Related Work . . . 12

3 Revisiting reopened bugs in open source software projects 19
- 3.1 Introduction . . . 20
- 3.2 Data Collection . . . 26
- 3.3 Case Study Setup and Results . . . 29
- 3.4 Discussion . . . 55
- 3.5 Threats . . . 59
- 3.6 Chapter Summary . . . 60

4 Studying post release reopened bugs in open source software projects 62
- 4.1 Introduction . . . 63
- 4.2 Data Collection . . . 70
- 4.3 Case Study Setup and Results . . . 75
- 4.4 Discussion . . . 98

4.5 Threats 107
4.6 Chapter Summary 110

5 Conclusions and Future Work **112**
5.1 Thesis Contributions 113
5.2 Future work 114

Bibliography **115**

List of Tables

3.1 The features that we use in our reopened bug prediction pipeline 28
3.2 Overview of the used hyperparameters in our study. The default values are shown in bold. Randuniform denotes drawing random samples from a uniform distribution, and intranduniform denotes denotes drawing random samples from a uniform distribution and converting them to their nearest integer value. 36
3.3 Percentage of projects with which the constructed reopened bug prediction pipeline with different learners yields an acceptable AUC 39
3.4 The distribution of major field changes that occur in reopened bugs grouped by before, during, and after reopening a bug. 47
3.5 Different reasons for reopening the bugs based on various rationale categories (i.e., before bug reopening, during bug reopening and after bug reopening 49
3.6 Different reasons for reopening the bugs identified before bug reopening in various project categories (i.e., acceptable, and poor) 54
3.7 Our major findings and their implications. 56

4.1 The features that we use in our prediction pipeline 72
4.2 Distribution of the types of release based reopened bugs 80
4.3 Overview of the used hyperparameters in our study. The default values are shown in bold. Randuniform denotes drawing random samples from a uniform distribution, and intranduniform denotes denotes drawing random samples from a uniform distribution and converting them to their nearest integer value. 93
4.4 Top three feature importance ranks for various learners (i.e., classifiers) to predict if a post-release reopened bug will get resolved rapidly/slowly 97
4.5 Top three feature importance ranks for various learners (i.e., classifiers) to predict if a pre-release reopened bug will get resolved rapidly/slowly 102
4.6 Our major findings and their implications. 103

List of Figures

2.1 The workflow of the bug fixing process 9
2.2 Bug report (id: 4234), bug report (id: 1403) without the *affected version* field set for the Apache Zookeeper project, and with the *affected version* field set for the Apache Spark project respectively. 11

3.1 Historical events in a reopened bug 21
3.2 Workflow of data collection and pre-processing 30
3.3 Reopened bug prediction pipeline 32
3.4 Caption for LOF 41

4.1 A post-release reopened bug report (id: 2325) for the Apache Zookeeper project. 66
4.2 Workflow of data collection and preprocessing 74
4.3 Classifying a bug report into pre and post-release reopened bugs. 75
4.4 Distribution of the time spent in various scenarios for pre-release and post-release reopened bugs. 82
4.5 Distribution of the difference between the time spent from first bug reopening to final bug resolving and time spent from bug creation to first bug resolving. 83
4.6 Distribution of bug rework time for various reasons of reopening a bug in pre-release and post-release reopened bugs. 85
4.7 A model to predict if a post-release reopened bug will get resolved rapidly/slowly. 88
4.8 Distribution of AUC for various learners (i.e., classifiers) and median AUC scores to predict if a post-release reopened bug will get resolved rapidly/slowly. 95
4.9 Distribution of AUC for various learners (i.e., classifiers) and median AUC scores to predict if a pre-release reopened bug will get resolved rapidly/slowly 101

CHAPTER 1

Introduction

BUG fixing is an important activity in software development. Normally, a bug is resolved and closed after it is fixed. However, sometimes it needs to be reopened. Reopened bugs are considered a disappointment to the end users in software development since it is advertised that a particular bug is fixed, but in reality, it is otherwise, so it needs to be reopened and reworked [64]. Mi and Keung [64] observed that around 6–10% of the bugs are reopened in four open source projects from the Eclipse product family. If a fair number of fixed bugs get reopened, it is an indication of instability in the software project [120]. Reopened bugs consume more time to resolve than normal bugs by a factor of 1.6–2.1 [64] and cause extra rework time for development teams [65]. Reopened bugs also lead to a loss of end-users trust regarding the quality of the software [83]. In particular, reopened bugs that are reopened

post-release are even more critical as they impact a larger audience (i.e., clients).

Hence, many prior studies proposed models to predict if a bug will be reopened. For example, Shihab et al. [84] studied the reopening likelihood of a bug based on four dimensions of features (i.e. work habits, bug report, bug fix, and team dimensions). Xia et al. [105] proposed the ReopenPredictor[1] and improved the performance of the reopened bug prediction pipeline using the same dataset as used by Shihab et al. Many follow up studies on reopened bugs focused only on the Eclipse project, while there is a dramatic increase in the number of open source projects and their associated bugs over the years. Therefore, it becomes important to understand reopened bug prediction on a larger scale so that the findings can be generalized.

Following the above understanding of the characteristics of reopened bugs, it is also important to consider reopened bugs in the context of the life cycle of a software system. A bug is considered as a pre-release bug for a software release version if it is identified before the release of that particular version, whereas a bug is considered as a post-release bug for a software release version if it is identified after the release of that particular version. Reopened bugs can be categorized into two categories based on the software release (i.e., pre-release reopened bugs and post-release reopened bugs). Pre-release bugs that are reopened at least once are considered as pre-release reopened bugs and post-release bugs that are reopened at least once are considered as post-release reopened bugs. Rwemalika et al. [78] observe that post-release bugs need at least two times bigger chunks of changes as opposed to pre-release bugs and 66.7% of the post-release bugs require changes in configuration files as compared to only 46% configuration file changes in case of pre-release bugs. Therefore, it is important to closely examine post-release bugs. Traditional studies on defect prediction consider

[1]https://github.com/xin-xia/reopenBug

segregating bugs into pre-release and post-release bugs for their analysis [49, 50, 80]. However, prior studies on reopened bugs [64, 83, 84, 105, 106] focused on reopened bugs as a whole (i.e., without factoring when a bug was reopened in the life cycle of a software system). It is also important to determine if a release based reopened bug will get resolved rapidly/slowly. Therefore, we determine factors that lead to post-release reopened bugs getting resolved rapidly. These insights can help software developers to handle post-release reopened bugs efficiently.

1.1 Thesis Statement

Reopened bugs lead to a great loss in the quality of the software system. Hence, it is important to understand the time and release related characteristics of reopened bugs to avoid their occurrence and speed up their closing.

1.2 Thesis Overview

In this section, we provide an outline of our thesis.

1.2.1 Chapter 2: Background and Related Work

In this chapter, we discuss the background of reopened bugs. We provide an overview of reopened bugs in open source systems. We also discuss prior work in this field. In particular, we focus on related work in the following areas: 1) study of reopened bugs characteristics 2) empirical studies of reopened bugs 3) modelling reopened bugs.

1.2.2 Chapter 3: Revisiting reopened bugs in open source software systems

Reopened bugs consume almost 1.6–2.1 more time to resolve than normal bugs. Prior studies on reopened bugs constructed prediction pipelines to determine the likelihood of a bug getting reopened. Prior research focused on three projects (i.e. Apache, Eclipse, and OpenOffice) for their analysis. When we examined the bug reports from the data shared by prior studies, we observed that one out of the three projects (i.e., Apache project) has a data leak issue. Therefore, research on predicting the likelihood of a bug getting reopened is effectively based on only two projects and hence can't be generalized. Moreover, all the prior studies used an old prediction pipeline and with time, such a pipeline has improved. For instance, no prior study tuned the hyper-parameters of their studied model, nor did they perform correlation and redundancy analysis on the independent features of their dataset to remove correlated features. Therefore, in this chapter, we study reopened bugs using a large scale dataset consisting of 47 projects using the latest prediction pipelines.

Using the updated pipeline across the 47 studied projects, we are able to achieve an acceptable AUC ($\geqslant 0.7$) on 34% (16/47) of the projects. Moreover, reopened bug prediction pipelines constructed with Random Forest and Gradient Boosting learners yield an acceptable AUC ($\geqslant 0.7$) on more studied projects than the prediction pipeline constructed with other learners. In addition, we observe that *number of comments* is the most important feature that helps in predicting the reopening likelihood of a bug which contrasts with findings of prior studies which found that the *comment text* is the most important feature in determining the likelihood of reopening a bug. Furthermore, we identify four categories of reasons for reopening a bug, that is, 1) technical

(i.e., patch/integration issues), 2) documentation (i.e., updating fields of a bug report), 3) human (i.e., incorrect assessment by a developer), and 4) no stated reason in a bug report. Our study provides new and more general insights into bug reopening activities. We encourage future research to leverage the rich data available during and after a bug is reopened, while this data was not previously used in understanding reopened bugs. We further encourage researchers to explore the management of reopened bugs based on the whole history of a bug to understand their eventual resolution.

1.2.3 Chapter 4: Studying post-release reopened bugs in open source software systems

Rwemalika et al. [78] observe that post-release bugs need at least two times bigger chunks of changes as opposed to pre-release bugs and 66.7% of the post-release bugs require changes in configuration files as compared to only 46% configuration file changes in case of pre-release bugs. Therefore, it is important for the research community to focus on post-release bugs. However, in the context of reopened bugs, no such segregation of bugs into pre-release and post-release reopened bugs is studied by any prior study.

At best we are able to predict if a post-release reopened bug will get resolved rapidly/slowly with an AUC 6% greater than in the case of pre-release reopened bugs. We observe that the median time taken for closing a reopened bug in case of pre-release reopened bugs is almost twice more than post-release reopened bugs. We observe that the time required to resolve pre and post-release reopened bugs measured by time is dramatically different. Studying reopened bugs as a whole leads to biased implications. Therefore, we encourage researchers to consider pre-release and post-release

reopened bugs separately in their analysis.

1.3 Thesis Contributions

In this thesis, we study reopened bugs using a large scale dataset. We revisit and improve prior reopened bug prediction pipelines. Furthermore, we analyze reopened bugs throughout the lifecycle of a software system and build a prediction pipeline for identifying post-release reopened bugs that can be eventually fixed in a short time. Our main contributions in this thesis are as follows:

1. Large scale study of reopened bug characteristics with two case studies of reopened bug prediction and fixing time prediction for post-release reopened bugs.

2. We improve the prior reopened bug prediction pipeline and demonstrate that only 34% projects give an acceptable AUC ($\geqslant 0.7$) when predicting reopened bugs.

3. We find that for pre-release reopened bugs the median closing time is twice as much as post-release reopened bugs indicating that pre-release and post-release reopened bugs have different characteristics.

4. We observe that our models for predicting if a reopened bug will get resolved rapidly for post-release reopened bugs gives 6% greater AUC than AUC obtained for the same model using pre-release reopened bugs. Therefore, studying reopened bugs as a whole leads to biased implications. We encourage researchers to consider pre-release and post-release reopened bugs separately in their analysis.

CHAPTER 2

Background and Related Work

In this chapter, we first discuss the background of reopened bugs. We then discuss prior studies that are related to reopened bugs. We mainly focus on 3 streams of prior studies, these are 1) study of reopened bugs characteristics, 2) empirical studies of reopened bugs, 3) modelling reopened bugs.

2.1 Background

2.1.1 Bug tracking systems

Bug tracking systems manage various bug fixing activities. Bug tracking systems are used to create bug reports, where reporters can provide essential information about the symptoms and causes of bugs. Developers discuss potential fixes of the bug, update

the status of the bug in order for others to track the progress of the bug fixing process. Bugzilla[1] and JIRA[2] are two of the commonly used bug tracking systems.

Bugs from both Bugzilla and JIRA were studied in prior work. However, prior studies on reopened bugs used bugs from Bugzilla [64, 83, 84, 105, 106, 109]. To the best of our knowledge, no prior work has studied reopened bugs that are tracked by JIRA. For example, Shihab et al. studied reopened bugs in three projects, i.e., Apache, Eclipse, and OpenOffice, which are tracked by Bugzilla. Xia et al. [105] extended Shihab et al.'s study using more advanced techniques on the same Bugzilla dataset. Xia et al. [106] investigated the performance of various learners again on the same Bugzilla dataset.

2.1.2 Process of reopening a bug

A bug transitions through various statuses in its life cycle [46]. The relevant statuses are explained below:

- *Open:* When a new bug is created, its status is set as *open*.
- *In progress:* When a developer starts working on a bug, its status is set as *in progress*.
- *Resolved:* When a bug has been resolved, its status is set as *resolved*.
- *Closed:* When a bug has been fixed and it will not be worked upon anymore, its status is set as *closed*.
- *Reopened:* When a bug is not fixed, it needs to be reopened. Therefore, its status is set as *reopened*.

[1] https://www.bugzilla.org/
[2] https://www.atlassian.com/software/jira

Figure 2.1 shows the life cycle of a bug. Initially, a bug is created and its status is set as *open*. Then the bug is assigned to a developer. When the developer starts working on the bug, the bug status is set as *in progress*. When the bug is resolved, the bug's status is set as *resolved*. The bug is then verified and if the fix is correct then the bug status is set as *closed*. However, a bug can be reopened, and the bug status is set as *reopened*.

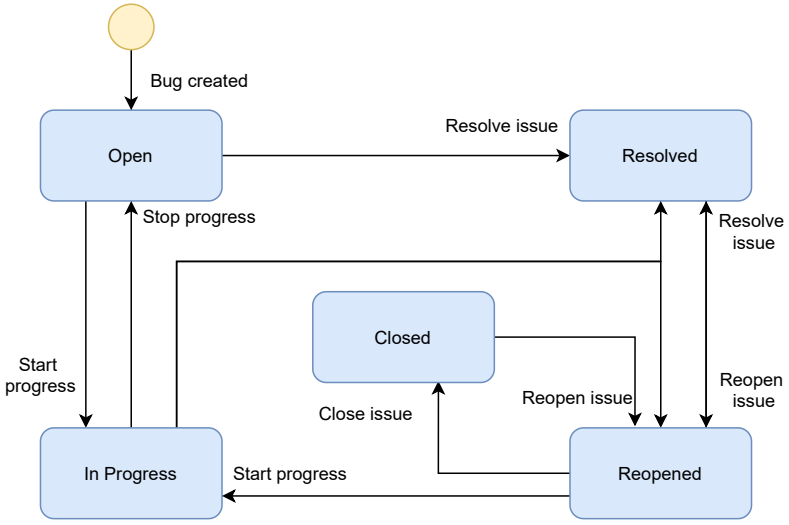

Figure 2.1: The workflow of the bug fixing process

Bug reopening can involve a discussion between developers. For example, in a bug report[3] (ID: 6089) for the Apache CXF project, a developer discusses with another developer that *"mmh... reopening as there's a testcase failing, I didn't notice that locally (likely run with -Pfastinstall) Rebecca, please have a look, I might revert the changes in the mean time."* [85]. In another example, a developer pointed out in a bug report[4] (ID: 737) for the Apache HAWQ project that the bug will be reopened to update the version [112].

[3] https://issues.apache.org/jira/browse/CXF-6089
[4] https://issues.apache.org/jira/browse/HAWQ-737

2.1.3 Tagging affected version of a bug

Software systems have multiple releases. Bugs are associated with particular version(s) of software releases. JIRA bug reports provide the *affected version* field in a bug report in order for developers to indicate the version(s) of the software system which is affected by that particular bug. Such details are important for software development as development teams can track the progress of bugs w.r.t. software release versions. Some bug reports contain the affected version information. However, in other bug reports the affected version is not tagged. Figure 2.2 shows bug reports with both scenarios. We observe that in a bug report[5](id: 4234) for the Apache Zookeeper project, the affected version is not indicated. There is no mechanism to determine if a bug report without an affected version is due to the bug not being related to any release version, or the developer forgetting to tag the affected version. Therefore, we consider a bug without the *affected version* field set as an *unidentified release*. In a bug report[6](id: 1403) for the Apache Spark project, multiple affected versions are indicated (i.e., 1.0.0, 1.3.0 and 1.4.0).

2.1.4 The various definitions of pre-release and post-release bugs

Prior studies on pre-release and post-release bugs relied on various definitions of pre-release and post-release bugs. Munaiah et al. [68] defined pre-release bugs as those that were fixed prior to a particular release. Chen et al. [20] considered bugs that were

[5] https://issues.apache.org/jira/browse/ZOOKEEPER-4234
[6] https://issues.apache.org/jira/browse/SPARK-1403

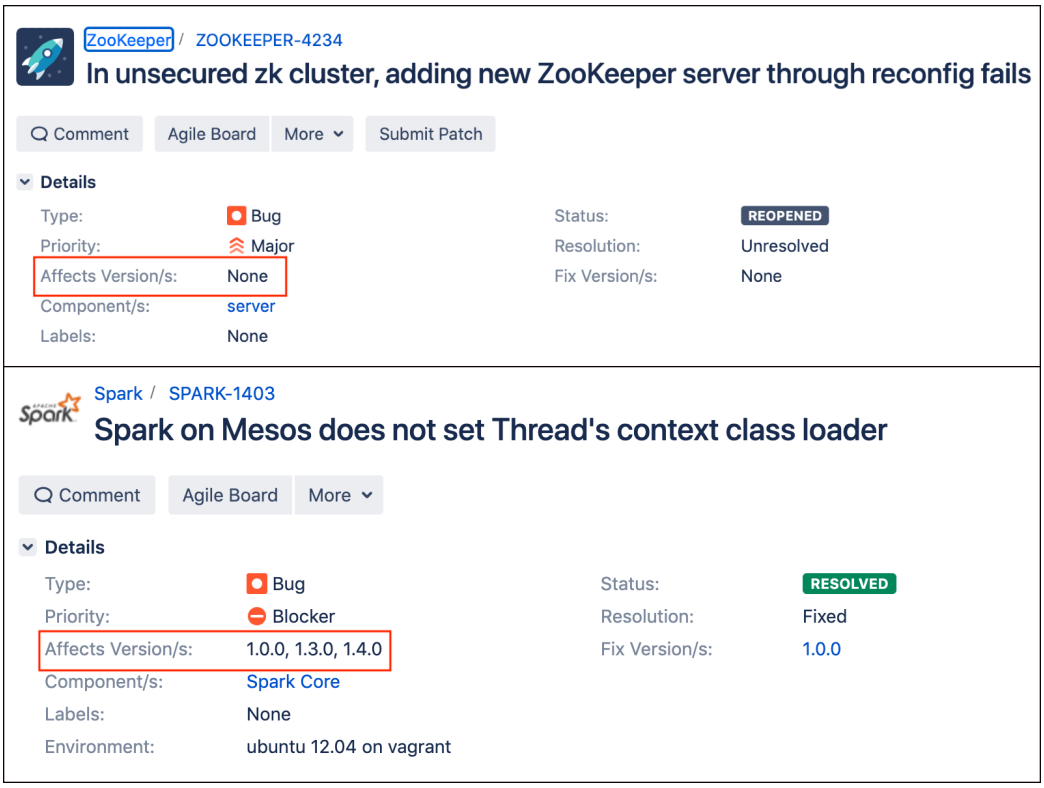

Figure 2.2: Bug report (id: 4234), bug report (id: 1403) without the *affected version* field set for the Apache Zookeeper project, and with the *affected version* field set for the Apache Spark project respectively.

found before the system is released as pre-release bugs. Multiple other studies considered issues that occur before a software release as pre-release bugs [80, 114]. Rwemalika et al. [78] defined a pre-release bug as a bug that is detected during software development. Da Costa et al.; Yatish et al.; Khomh et al. [23, 50, 113] observe that a bug can be associated with multiple releases and they use the date of the earliest affected version and compare that with the release date of those affected versions to determine if a bug is a post-release bug. Zimmermann et al.; Schroter et al. [80, 121] considered bugs that occur after the release of the system as post-release bugs. Rwemalika et al. [78] defined post-release bug as a bug that escaped to production.

A bug report has a field "affected version" that represents the software release version(s) affected by the particular bug. Prior studies (as discussed above) define both pre-release and post-release bugs using various definitions. However, it is more accurate to consider information about when the affected version is tagged to the bug to determine if a bug is a pre-release or a post-release bug. Moreover, a bug can be associated with multiple affected versions and hence can be a pre-release and a post-release by different affected versions. For example, in a bug report[7] (id: 1403) for the Apache Spark project, the bug is associated with three affected versions (i.e., 1.0.0, 1.3.0 and 1.4.0). The above mentioned bug is a pre-release bug w.r.t affected version 1.0.0 since, the affected version 1.0.0 was tagged on 03-April-2014, whereas the affected version 1.0.0 was released later on 30-May-2014[8]. However, this bug is a post-release bug for affected versions 1.3.0 and 1.4.0, since versions 1.3.0, and 1.4.0 were tagged on 11-June-2021 and 17-June-2015 respectively, and the version 1.3.0 and 1.4.0 were released earlier on 13-March-2015 and 11-June-2015 respectively. All prior studies consider a bug as either a pre-release or a post-release bug. However, it is more accurate to tag bugs as pre-release and post-release w.r.t particular affected version/s.

2.2 Related Work

2.2.1 Study of Reopened Bugs Characteristics

Prior studies focused on bug characteristics by studying various aspects of software bugs. Xia et al. [104] proposed a MTM (multi-feature topic model) for bug triaging. Bug

[7] https://issues.apache.org/jira/browse/SPARK-1403

[8] https://issues.apache.org/jira/projects/SPARK?selectedItem=com.atlassian.jira.jira-projects-plugin:release-page&status=released-unreleased

triaging refers to the performed activities by developers on un-assigned bugs, such as determining whether the bug needs attention if so assigning the bug to a developer, and adding a milestone for the bug [13]. Jalbert and Weimer [40] proposed a system to automatically identify duplicate bugs. They observed that their system is able to reduce software development costs by removing 8% of the duplicate bugs. Tian et al. [94] extended their work and improved the accuracy of duplicate bug detection. Somasundaram and Murphy [86] recommended a list of components that are most relevant to a bug. Xia et al. [108] found that most of the bugs set their priority and severity value as the default value. Prior studies also focussed on bug assignment tasks in the triaging process. Murphy and Cubranic [69] proposed machine learning algorithms using text categorization to predict the assignee of a bug. Other studies by Avnik et al.; Xia et al.; Tian et al. [6, 95, 107] proposed models to recommend the assignee for a bug. Our study also investigates bug management with respect to how bugs are reopened. We extend prior studies of bug reopening through a large scale study using a modern machine learning pipeline to investigate the generalizability of reopened bug prediction pipelines.

Bugs can be categorized into two types i.e., pre-release bugs and post-release bugs. A bug is considered as a pre-release bug if the affected version (release version) is tagged to the bug report before the release of that affected version. A bug is considered as a post-release bug if the affected version (release version) is tagged to the bug report after the release of that affected version. Rwemalika et al. [78] studied the characteristics of pre-release and post-release bugs. They analyzed 37 projects from BGL BNP Paribas. They found that post-release bugs are more complex to resolve, involving a median of 16.5 lines edited as compared to pre-release bugs with a median of 8.5 lines edited.

They also found that pre-release bug fixes involve more design related changes and post-release bug fixes involve more control flow modifications.

Jiang et al. [41] developed a release policy for software development on when to stop testing and release the product. Such a policy can help in reducing the overall cost and associated risk of bug. They propose a policy involving performing testing on post-release bugs in order to avoid giving up on substantial market opportunity costs.

Khomh et al. [50] compared the median uptime, crash rates, and percentage of post-release bugs that have a short release cycle. They empirically studied the development process of Mozilla Firefox releases in 2010 and 2011. They observed that with release cycles getting shorter, bugs are being experienced earlier during software execution, and bugs are fixed rapidly.

Herzig et al. [39] used test failure metrics to predict pre-release and post-release bugs in Windows 8 operating system projects. Herzig et al. notes that their prediction pipeline trained on test failure data of Windows 8 showed higher precision and recall as compared to previous pre-release and post-release prediction pipelines on Windows 8.

2.2.2 Empirical Studies of Reopened Bugs

Prior studies empirically mined bug reopening to gain a deeper understanding of reopened bugs. Shihab et al. [84] conducted an investigation of the bug resolving time for reopened and non-reopened bugs in the Eclipse project. They observed that reopened bugs take at least twice as much time to resolve than non-reopened bugs (i.e., on average of 149 days for non-reopened bugs and 371 days for reopened bugs). Guo et

al. [32] conducted an empirical study on reopened bugs in Microsoft Windows and observed that if a bug is reopened too many times then probably it will never be fixed. Mi et al. [65] discussed a new type of reopened bug that has no direct impact on the user experience or the normal operation of the system. They showed statistically that such reopened bugs are significantly different from other reopened bugs. Mi and Keung [64] studied four open source projects, i.e., CDT, JDT, PDE, and Platform, to quantitatively analyze reopened bugs in regards to their impacts, proportions, and evolution. They identified that 6% to 10% of the bugs are reopened and reopened bugs remain active between 1.6 and 2.1 times higher than non-reopened bugs.

2.2.3 Modelling Reopened Bugs

Prior studies leveraged machine learning models to predict various attributes in a bug report. Shihab et al. [83] conducted an initial study to predict reopened bugs in the Eclipse dataset using four dimensions and 22 features. Shihab et al. [84] also extended their study to three additional datasets (Apache, Eclipse, and OpenOffice). They built a Decision Tree model and performed a top node analysis to identify the most important features in predicting reopened bugs, and observed that *comment text* is the most important feature in predicting reopened bugs in the Eclipse project. Xia et al. [106] investigated the performance of various learners to predict reopened bugs. Among the 10 learners, they identified that Bagging and Decision Tree learner gives the best performance. Xia et al. also showed that their results outperform the prior work. Xia et al. [105] proposed the ReopenPredictor[9] tool to predict reopened bugs using the same dataset as used by Shihab et al. [84]. Xia et al. proposed a feature selection method

[9]https://github.com/xin-xia/reopenBug

for choosing the most substantial textual features (*comment text* and *description text*) for the imbalanced dataset from both reopened and non-reopened bugs. Xuan et al. [110] modeled the developer prioritization of the bugs to predict the reopening of a bug. Xuan et al. [109] discussed the benefits of data reduction techniques in reopened bug analysis. Caglayan et al. [15] studied a bug activity dataset in a large-scale enterprise software product. They analyzed four dimensions (i.e., issue-code relation, issue proximity network, issue reports, and developer activity) as possible factors that may cause a bug to be reopened. Zimmermann et al. [120] conducted a study to find the root cause of reopened bugs and built a model to predict reopened bugs. Tu et al. [96] defined data leak as information from the future used in predicting events making the prediction pipelines misleadingly optimistic. They examined the prior literature in prediction tasks related to bugs and found that 11 out of 58 studies have data leak issues.

Prior studies on reopened bug prediction pipelines focused only on a small dataset (i.e., three projects). Moreover, we identified that the Apache project used in the prior studies has a data leak issue. Instead, in this thesis, we use 47 projects to study reopened bugs. We also use 7 learners in our study with the latest techniques for generating a reopened bug prediction pipeline. Moreover, we also conduct a qualitative study to better understand the reasons why bugs get reopened.

The time it takes to finally resolve a bug from when a bug is created is known as bug lifetime. Predicting bug lifetimes can aid software developers to prioritize their work, better realize project timelines, and efficiently allocate resources [76]. Prior studies focused on predicting the lifetime of a bug. Rees-Jones et al. [76] studied 10 projects and a total of 52,670 bugs from JIRA. They used decision tree learner to build prediction

pipeline and used 5 time slices as labels, these are 1 day, 7 days, 14 days, 30 days, and 90 days. They recommend using a cross-project dataset for predicting bug lifetime. Panjer [72] studied Eclipse Bugzilla dataset to predict bug lifetime. Panjer [72] used dataset only available at the creation of the bug (ignoring dynamic dataset generated during operations on the bug report) to predict bug lifetime. Panjer divided the time to resolve into 7 time slices i.e., 1.4 days, 3.4 days, 7.5 days, 19.5 days,52.5 days,156 days, and more than 156 days. His model can predict bug lifetime only with 34.9% accuracy using Logistic Regression learner. Giger et al. [31] Constructed machine learning models to predict whether a bug will be fixed fast or take more resolution time using decision tree learner. Giger et al. studied 6 projects namely Eclipse JDT, Mozilla Core, Gnome GStreamer, Eclipse Platform, Mozilla Firefox, and Gnome Evolution. Giger et al. divided the time to resolve into 5 time slices i.e., 1 day, 3 days, 1 week, 2 weeks, 1 month. Giger et al. achieved an AUC of 0.74 for predicting fast fixed bugs in the Eclipse project.

There are other studies that predict the time taken to finally resolve a bug. All these studies take different fixing time slice thresholds. Kikas et al. [51] built models on 4,000 Github projects to predict whether or not a bug will get closed within a given time from various time spent points in a bug lifecycle. They use static as well as dynamic features in their study. They divided the time to resolve into 7 time slices i.e., 1 day, 7 days, 14 days, 30 days, 90 days, 180 days, and 365 days. They produced a model with a maximum AUC of 0.69 from the observation time (point in bug lifetime when the prediction is made) of 180 days. They found that observing an issue by extending the bug life cycle increases the predictive power of the model. Their study highlights the importance of dynamic and contextual features in fixing time prediction pipelines. Marks et al. [59]

used Random Forest learner to predict the fix-time for bugs. They divided the time to resolve into three time slices i.e., < 3 months, < 1 year, and < 3 years. They observed that using the Random Forest learner, they can correctly predict the fixing time of a bug with a success rate of 65%.

In our chapter 4, we predict whether a pre/post-release reopened bug will get resolved rapidly/slowly. We use all projects hosted on JIRA as our studied dataset. Our approach is different from prior work in the following ways 1) We consider the time spent between when a bug is first reopened and when the bug is finally resolved, whereas prior studies considered the time spent from bug creation to bug getting finally resolved. 2) We consider two time slices based on the time taken to resolve bug. Bugs in the top 20% and bottom 20% of fix time are labeled as fast resolved and slow resolved respectively, whereas prior studies considered time slices with predefined ranges.

CHAPTER 3

Revisiting reopened bugs in open source software projects

REOPENED bugs cause a loss of trust in the end-users regarding the quality of a software system. Hence, predicting bugs that might be reopened could be extremely helpful for software developers to avoid rework and avoid any loss of trust. Shihab et al. conducted a study to predict reopened bugs using three open source projects (i.e., Apache, Eclipse, and OpenOffice). Xia et al. further improved the performance of reopened bug prediction pipelines using the same dataset as used by Shihab et al. Moreover, other studies on reopened bug prediction focus only on the Eclipse project, while open source projects and their associated reopened bugs have dramatically increased over the years. In general, the insights from the prior reopened bug studies were based only on three open source projects, and we observe that one out of the three projects (i.e., Apache) even has a data leak issue – the bug status of

reopened was included as training data to predict reopened bugs. In addition, prior studies used an outdated prediction pipeline (i.e., with old techniques for constructing a prediction pipeline) to predict reopened bugs. Therefore, we revisit the reopened bugs study on a large scale dataset consisting of 47 projects tracked by JIRA using a modern prediction pipeline that makes use of modern techniques such as SMOTE, and permutation importance together with 7 different machine learning models. We study reopened bugs using a mixed methods approach (i.e., we conduct both a quantitative and qualitative study). We find that: 1) Using the modern pipeline, only 34% projects give an acceptable performance with AUC $\geqslant$ 0.7. 2) We identify four major reasons for a bug getting reopened, that is, technical (i.e., patch/integration issues), documentation, human (i.e., due to incorrect bug assessment), and reasons not shown in the bug reports. 3) In projects with an acceptable AUC, 94% of the reopened bugs are due to patch issues (i.e., the usage of an incorrect patch). We encourage future research to leverage the rich data available during and after a bug is reopened, as such data was not previously used in understanding reopened bugs.

3.1 Introduction

Fixing bugs is an important part of software development. Generally, a bug is resolved and closed after the bug is fixed. However, sometimes a bug needs to be reopened. Mi and Keung [64] observed that around 6–10% of the bugs are reopened in four open source projects from the Eclipse product family. If a fair number of fixed bugs get reopened, it is an indication of instability in the software project [120]. Reopened bugs consume more time to resolve than normal bugs by a factor of 1.6–2.1 [64]. Reopened bugs cause extra rework effort for development teams [65]. Moreover, reopened bugs

also lead to a loss of end-users trust in the quality of the software [83]. Figure 3.1 shows a reopened bug[1]. In this bug, initially, the bug was non-reproducible; however, later the developer reopened the bug. The bug was in the *reopened* state for more than three years without any activity and later the bug was labeled as *won't fix*.

Figure 3.1: Historical events in a reopened bug

Therefore, to avoid the reopening of a bug, prior studies proposed models to predict if a bug will be reopened (i.e., reopened bug prediction pipelines). For example, Shihab et al. [84] studied the reopening likelihood of a bug based on four dimensions of features (i.e. work habits, bug report, bug fix, and team dimension). They studied three projects (i.e. Apache, Eclipse, and OpenOffice) using 44,317 non-reopened bugs and 11,745 reopened bugs. Xia et al. [105] proposed the ReopenPredictor[2] and

[1] https://issues.apache.org/jira/browse/MAPREDUCE-6378
[2] https://github.com/xin-xia/reopenBug

further improved the performance of the reopened bug prediction pipeline using the same dataset as used by Shihab et al. Moreover, other studies on reopened bug prediction focused only on the Eclipse project, while there is a dramatic increase in the number of open source projects and their associated bugs over the years. However, when we examined the bug reports from the data shared by Shihab et al. and Xia et al., we observed that one out of the three projects (i.e., Apache project) has a data leak issue [48, 96] – the bug status of *reopened* was included as training data to predict reopened bugs. Therefore, if we discount the findings from the Eclipse project, the findings of the prior studies are effectively based on only two projects (i.e., Eclipse, and OpenOffice). Furthermore, all these prior studies used an old prediction pipeline and with time, the techniques for prediction pipelines have improved. For instance, no prior study in reopened bug prediction tuned the hyperparameters of their studied model, which could, in turn, improve the prediction performance of the constructed reopened bug prediction pipelines [74, 90]. No prior study in reopened bug prediction performed correlation and redundancy analysis on the independent features of their dataset to remove correlated features; however, several recent studies have used correlation and redundancy analysis on their dataset prior to training the prediction pipeline [24, 43, 54, 60, 73, 74, 90]. This motivated us to perform a large scale study of the likelihood of reopening a bug using a modern prediction pipeline. In our study, we answer the following research questions.

RQ1: How well do bug reopen models generalize?

Motivation: Prior studies on determining the likelihood of reopening a bug by Shihab et al. [84] and later by Xia et al. [105] were based only on three projects (i.e., Apache, Eclipse, and OpenOffice). Moreover, the other two studies by Shihab et

al. [83] and by Xia et al. [106] on reopened bug prediction were based only on one project (i.e., Eclipse). While examining their used dataset, we observe that out of their three studied projects, one project (i.e., Apache) has a data leak issue, implying that the prior findings on reopened bugs prediction are effectively based on only two projects (i.e., Eclipse, and OpenOffice). In addition, all these studies were conducted 6–10 years ago, while open source projects are growing in the meanwhile. Furthermore, several experimental components (such as the use of Decision Tree learner, class re-weighting and re-sampling strategy for handling class imbalance, and top node analysis for generating feature importance of the prediction pipeline) that prior studies used to build a reopened bug prediction pipeline have since then been shown to be not as effective. The use of the latest techniques, such as hyper-parameters tuning, correlation and redundancy analysis, SMOTE for handling class imbalance, multiple models, feature encoding, and permutation importance analysis for generating feature importance, are found to be more effective for building a prediction pipeline [2, 5, 27, 43, 60, 90, 91]. Moreover, after Shihab et al. and Xia et al.'s studies, no subsequent study revisited the reopened bug prediction problem with these pipeline improvements in mind. Therefore, in this RQ, we revisit the prior findings of reopened bug prediction through a large scale study of 47 new projects that we collect from JIRA as we outline in Section 3.2 and with the latest techniques for building prediction pipelines to determine the likelihood of a bug being reopened.

Results: Using the updated pipeline, we observe that across the 47 studied projects, at best we are able to achieve an acceptable AUC ($\geqslant 0.7$) on 34% (16/47) of the

projects. Furthermore, reopened bug prediction pipelines constructed with Random Forest and Gradient Boosting learners yield an acceptable AUC ($\geqslant 0.7$) on more studied projects than the model constructed with other learners. Finally, with our large scale study, we observe that *number of comments* is the most important feature in predicting the reopening likelihood of a bug. Such a finding contrasts with the findings observed by Shihab et al. where they found that the *comment text* is the most important feature in determining the likelihood of reopening a bug.

RQ2: Why do bugs get reopened?

Motivation: Zimmermann et al. [120] observe that understanding bug reopening helps in understanding various software development tasks such as identifying areas that need better tool support and improving the bug triage process. From the findings in RQ1, we observe that 34% of our studied projects can be leveraged with confidence for providing insights about reopened bugs. In addition, we observe that interpreting the reopened bug prediction pipeline constructed on projects with an acceptable AUC, the generated insights were coarse. For example, by interpreting the reopened bug prediction pipeline on the dataset with an acceptable performance, we could only identify that *number of comments* and *comment text* as the top two most important features in determining the likelihood of reopening a bug. However, such insights do not provide any actionable suggestions that can be used to either prevent bug reopening or identify the potential root cause of reopened bugs. Moreover, we observe that some developers pointed out the rationale

of why a bug was reopened when they reopened the bug. For example, in a bug report[3] (ID: 4112) for the Apache Accumulo project, the bug was considered as non-reproducible initially and was resolved but later a developer reopened the bug and at the same time noted that *"Reopening, because the changes for this should either be reverted or another solution should be applied."* Therefore, in this RQ, we investigate the rationale behind why bugs are reopened by leveraging the whole history of a bug report. In addition, we investigate if the reasons for reopening a bug differ between the projects on which the constructed reopened bug prediction pipelines have an acceptable versus poor performance.

Results: Through a mixed methods approach, we identify that in 63.1% and 86.7% of the reopened bugs, developers post comments during and after the reopening respectively, and in 77.6% of the reopened bugs, the reason to reopen a bug is identified either during or after the bug reopening. Through a qualitative study of 370 samples of reopened bugs (with a 95% confidence level and a 5% confidence interval), we identify four categories of reasons for reopening a bug, that is, technical (i.e., patch/integration issues), documentation (i.e., updating fields of a bug report), human (i.e., incorrect assessment by a developer), and no stated reason in the reopened bug report. In addition, through our qualitative study, we observe that in projects with an acceptable AUC, 94% of the reopened bugs are due to patch issues and a negligible proportion of 4% bugs are reopened due to integration issues, whereas in projects with a poor AUC, apart from patch issues (66%), bugs are reopened due to integration issues (22%).

Based on our findings, we encourage developers to carefully examine their patches

[3] https://issues.apache.org/jira/browse/ACCUMULO-4112

for bug fixing, e.g., by testing the patches thoroughly before closing the bugs, as the most common reasons for reopening bugs are technical issues (i.e., patch/integration issues). Also, we suggest that bug tracking systems should enable developers to edit certain fields of bug reports without requiring the reopening of a bug, as 24.3% of our manually studied reopened bugs are reopened due to documentation (bookkeeping) issues.

The rest of this chapter is organized as follows. Section 3.2 describes the data collection processes. Section 3.3 discusses our case study. Section 3.4 presents a discussion about our findings. Section 3.5 presents the threats to the validity of our study. Finally, Section 3.6 discusses the conclusion of this chapter.

3.2 Data Collection

We describe our data collection process in this section. For revisiting the reopened bugs study on a large dataset, we collect 47 projects from JIRA (we refer to them as the JIRA dataset) and map all the features of the Bugzilla dataset that are also present in the JIRA dataset. In Section 3.2.1, we describe the data collection processes of the JIRA dataset. The JIRA dataset contains categorical (i.e., nominal), numeric, boolean, and text features. We discuss these features in detail in Section 3.2.1, and in Section 3.2.2 we discuss our process to clean the collected data.

3.2.1 Collecting the JIRA dataset

We study the reopened bugs that are tracked by JIRA. We use the JIRA REST API[4] to collect the JIRA bugs. We collected all the bugs that were ever reopened across all projects available in JIRA. For non-reopened bugs, we collected all the bugs which were either closed or resolved but were never reopened. In total, we collected 388,404 non-reopened bugs and 25,376 reopened bugs across 589 projects. All the features except the *number of files in the fix* were collected using the JIRA REST API. We used Pydriller[5] to collect the number of changed files in the corresponding GitHub repositories using the approach outlined by Vieira et al. [101]. Table 3.1 shows the 20 features that we collected for our study. Note that the features (i.e., *platform*, *severity*, *severity changed* and *number in the CC list*) are not present in the JIRA. We further filter the studied projects using two criteria: 1) we select the top 50 projects in JIRA based on the committers count[6], 2) We remove the projects that do not have at least 10 reopened bugs. By applying these two criteria, we end up with 47 projects to study (including 178,636 non-reopened bugs and 9,993 reopened bugs). While computing the value of features, we ignore all events in the bug report that occur at the time of reopening the bug or after reopening the bug, to prevent any data leak issue. For example, for the *comment text* feature, we only considered those comments that were posted before the reopening of each studied bug.

[4] https://developer.atlassian.com/cloud/jira/platform/rest/v2/intro/
[5] https://pydriller.readthedocs.io/en/latest/
[6] https://projects.apache.org/projects.html?number

Table 3.1: The features that we use in our reopened bug prediction pipeline

Category	Shihab et al. features	Our features (JIRA)	Type	Explanation
Work habits	Time	Time	Nominal	Time (morning, afternoon, evening, night) at the time of first closing of the bug.
	Weekday	Weekday	Nominal	Day of the week at the time of first closing of the bug.
	Month day	Month day	Numeric	Day of the month at the time of first closing of the bug.
	Month	Month	Numeric	Month at the time of first closing of the bug.
	Day of year	Day of year	Numeric	Day of year at the time of first closing of the bug.
Bug report	Component	Component	Nominal	The component of the bug.
	Platform	-	-	-
	Severity	-	-	-
	Priority	Priority	Numeric	The priority of the bug.
	Number in the CC list	-	-	-
	Description size	Description size	Numeric	The number of words in the description.
	Description text	Description text	Text	The text of the description.
	Number of comments	Number of comments	Numeric	The number of comments in the bug.
	Comment size	Comment size	Numeric	The number of words in the comments.
	Comment text	Comment text	Text	The text of the comments.
	Priority changed	Priority changed	Boolean	Whether priority changed.
	Severity changed	-	-	-
Bug fix	Time days	Time days	Numeric	The time it took to close the bug.
	Last status	Last status	Nominal	The last status when the bug was first closed.
	Number of files in the fix	Number of files in the fix	Numeric	The number of files in the fix.
People	Reporter name	Reporter name	Nominal	The name of the reporter of the bug.
	Fixer name	Assignee name	Nominal	The name of the assignee of the bug.
	Reporter experience	Reporter experience	Numeric	The number of bugs that the reporter has reported before reporting this bug.
	Fixer experience	Assignee experience	Numeric	The number of bugs assigned to the assignee before this bug.

3.2.2 Pre-processing features

Figure 3.2 shows the steps we followed to collect bug reports and how we process the data. Our dataset contains two text features, i.e., the *description text* and *comment text*. We pre-processed the text features similar to prior studies [25, 47, 98]. The comments and description of a bug report contain URLs. We replace them with a placeholder token to simplify the representation of URLs [38]. Developers add code blocks in the description or comments of a bug report. To simplify our text pre-processing, we replace code blocks with a placeholder token. Error logs and stack traces are processed similarly, i.e., by replacing them with a placeholder token. We then remove all the stop words as they do not contribute much to the context of reopened bugs [87]. We then use the NLTK package[7] to stem the text to reduce the different forms of each word in a bug report [34, 62, 88]. Finally, we use the pre-processed text in our reopened bug prediction pipeline together with the other features that are categorical (e.g., *component*, *last status*, and *assignee*), numeric (e.g., *month*, *description size*, and *number of comments*), or boolean (e.g., *priority changed*).

3.3 Case Study Setup and Results

In this section, we describe our case study results based on our new dataset to study reopened bugs. We first use the new JIRA dataset to predict reopened bugs (in Section 3.3.1). Then we analyze the studied bugs to understand the rationale for reopening a bug report (in Section 3.3.2). For each RQ, we provide the motivation, approach, and results of the RQ.

[7] https://www.nltk.org/

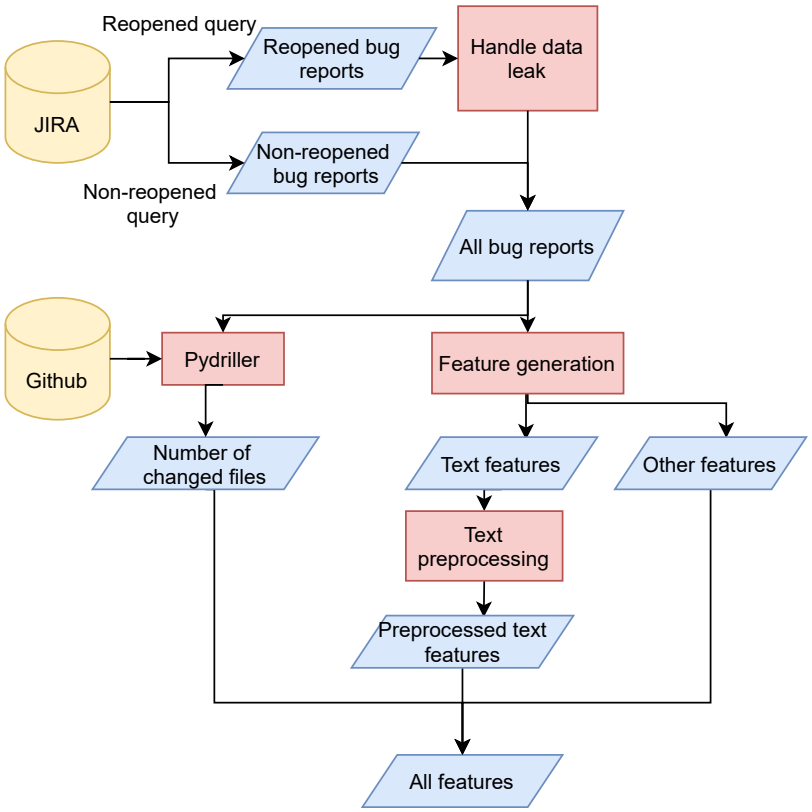

Figure 3.2: Workflow of data collection and pre-processing

3.3.1 RQ1: How well do bug reopen models generalize?

Motivation: Prior studies on determining the likelihood of reopening a bug by Shihab et al. [84] and later by Xia et al. [105] were based only on three projects (i.e., Apache, Eclipse, and OpenOffice). Moreover, the other two studies by Shihab et al. [83] and by Xia et al. [106] on reopened bug prediction were based only on one project (i.e., Eclipse). After examining the dataset used by these prior studies, we observe that out of their studied three projects, one project (i.e., Apache) has a data leak issue – the bug status of *reopened* was included as training data to predict reopened bugs. Hence, prior findings on determining the likelihood of reopening a bug were effectively based only

on two projects. In addition, these prior studies were conducted 6–10 years ago. Several experimental components that the prior studies used to build a prediction pipeline have since then been shown to be not as effective when building a prediction pipeline. For example, prior studies on reopened bug prediction did not tune the hyperparameters of their used model. However, several recent studies show that tuning the hyperparameters of a model is pivotal to ensure its optimal performance and interpretation [27, 90, 91]. Furthermore, prior studies [84] used a class re-weighting and re-sampling strategy to deal with the class imbalance issue that is present in the projects. However, several recent studies have shown SMOTE (Synthetic Minority Over-sampling Technique) to be a more effective method for class re-balancing [2, 90]. Prior studies [84] used top node analysis with a Decision Tree learner for generating feature importance of their model; however, recent studies show that the results of top node analysis can be biased since Decision Tree favors categorical features [5]. Moreover, prior studies [84, 105] did not use feature encoding with the latest and more widely used encoding techniques [17] such as one-hot encoding to encode the categorical features.

After Shihab et al. and Xia et al.'s study, no subsequent study revisited the reopened bug prediction problem with these pipeline improvements in mind. Therefore, in this RQ, we revisit their findings on a large scale study of 47 new projects that we collect from JIRA as we outline in Section 3.2 and with the latest techniques for building reopened bug prediction pipeline to determine the likelihood of bug reopening.

Approach: We construct our reopened bug prediction pipeline in this RQ. Figure 3.3 shows the prediction pipeline that we use to construct our reopened bug prediction pipeline. We discuss each step of the pipeline in detail below.

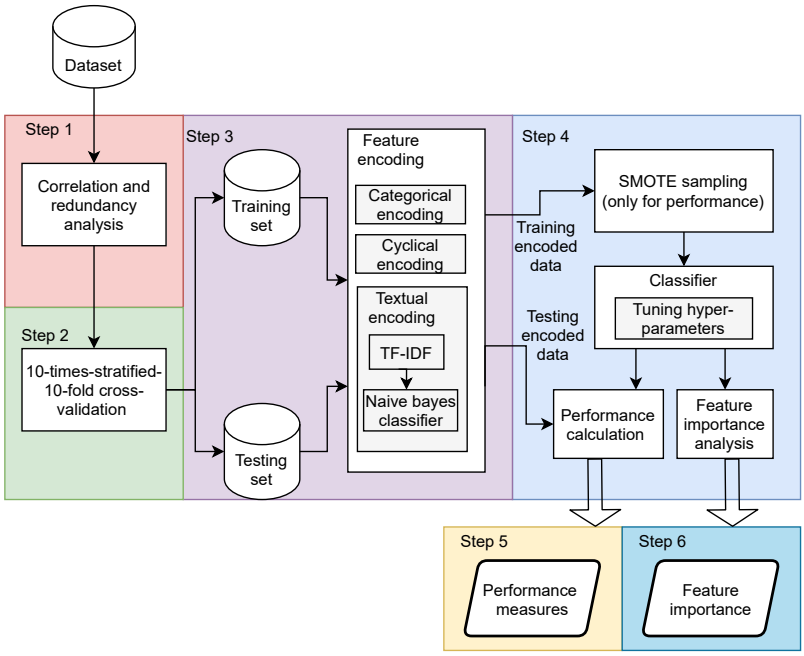

Figure 3.3: Reopened bug prediction pipeline

Step 1: Correlation and redundancy analysis. Tantithamthavorn et al. [90] outlines that presence of correlated and redundant features might result in affecting the computed feature importance ranks of a model. Therefore, similar to prior studies [24, 43, 54, 60, 73, 74, 90], we perform a correlation and redundancy analysis on the independent features of our dataset to remove correlated features using the implementation provided by Autospearman method of Rnalytica[8] R package [54, 90, 113]. We use a default value of ρ for the Spearman correlation coefficient threshold to remove the correlated features.

Step 2: Splitting the collected dataset into training/testing sets. We split the collected dataset into training and testing sets using 10-times-stratified

[8] https://rdrr.io/github/software-analytics/Rnalytica/man/AutoSpearman.html

10-fold cross-validation as opposed to 10-times 10-fold cross-validation used by Shihab et al. In stratified 10-fold cross-validation, the whole dataset is split into 10 folds in such a way that each fold contains the same percentage of all class labels as in the overall dataset. Each fold is then used to test the model performance and the remaining 9 folds are used for training the model. We repeat this process 10 times. Therefore, for each iteration 10 performance and feature importance measures are generated and an overall 100 performance and feature importance measures are generated on each studied project. For more details of the stratified k-fold cross-validation, please refer to Zeng et al. [116]. We choose 10-times-stratified-10-fold cross-validation in particular because our studied dataset is imbalanced, and stratified k-fold cross-validation helps preserve the original class distribution of the data [7, 36].

Step 3: Feature encoding.

Step 3.1 Categorical features encoding. We encode the categorical features in the dataset using one-hot feature encoding to transform categorical features present in the dataset into numeric features to be used as an input to train the model [119]. We employ one-hot feature encoding as opposed to simply converting the categorical features to numeric features as Shihab et al.; Cerda et al. [17] stated that one-hot feature encoding is one of the latest and more widely used categorical feature encoding procedure. One-hot encoding creates a separate boolean column for each category of the categorical features. These columns are then passed along with the other numeric columns to the next step.

Step 3.2 Cyclical features encoding. For the feature *weekday* we use cyclical encoding as feature *weekday* values are cyclic in nature [37].

Step 3.3 Textual features encoding. We use the TF-IDF (term frequency-inverse document frequency) vectorization [71] to encode the two text features (i.e., *comment text* and *description text*) that are present in the studied datasets similar to several prior studies [12, 61, 71, 75]. TF-IDF works by assigning a higher weight to a word if that word appears many times within a small number of documents. Similarly, TF-IDF assigns a lower weight to a word, when the given word appears fewer times in a document. TF-IDF assigns the lowest weight to a word if that given word appears in almost all the documents [21]. We use the training data in each iteration to build our TF-IDF model and use the trained TF-IDF model to generate the TF-IDF scores for text features in the testing data.

After transforming the text features with TF-IDF, we encode the text features as probabilities with a Naive Bayes learner similar to the prior study [63, 84, 117]. Because using only TF-IDF creates a column in the dataset for each word present in the dataset. Such an encoding might drastically increase the number of studied features and cause our reopened bug prediction pipelines to overfit the studied datasets. Furthermore, when we interpret the constructed reopened bug prediction pipelines, we are more interested in observing the importance of the studied text features (i.e., the *comment text* and *description text*) in predicting if a bug would be reopened (rather than the individual words). Therefore, we train two Naive Bayes learners on the training data.

Step 4: Model construction.

Step 4.1 Handling class imbalance for the reopened bug prediction pipeline. Since the dataset for determining the likelihood of reopening a bug is imbalanced (i.e., fewer reopened bugs as compared to non-reopened bugs), we used the SMOTE technique to rebalance the training dataset [58]. SMOTE is an oversampling approach in which the minority class is over-sampled by generating synthetic samples [18]. Shihab et al. used a class re-weighting and re-sampling strategy to deal with the class imbalance present in the dataset. However, we choose SMOTE for handling class imbalance as recent studies show that SMOTE is a more effective method for class re-balancing [2, 90].

Step 4.2 Training the model. We use the training set to train the reopened bug prediction pipeline. In our analysis, we use 7 commonly used learners in software analytics as mentioned in Table 3.2. We choose these learners in particular as a recent study by Agrawal et al. [1] notes, Decision Tree, Random Forest, Logistic Regression, Multinomial Naive Bayes, and K-nearest neighbors (KNN) are five of the most commonly used learners in software analytics. In addition, we also include Gradient Boosting and Adaboost learners as they are also commonly used in several software analytics studies [29, 90, 91].

Step 4.3 Tuning the hyperparameters of the learner. We use a grid search [81] based hyperparameters tuning to optimize the performance of the learner. Grid search exhaustively considers all combinations of parameter values used to train the model and chooses the parameter that yields the best

Table 3.2: Overview of the used hyperparameters in our study. The default values are shown in bold. Randuniform denotes drawing random samples from a uniform distribution, and intranduniform denotes denotes drawing random samples from a uniform distribution and converting them to their nearest integer value.

Learner	**Parameter name**	**Parameter description**	**Parameter range**
Random Forest	n_estimators	The number of trees in the forest.	(**100**, randuniform(50, 150))
	criterion	The function for measuring the split quality.	(**gini**, entropy)
	min_samples_split	The minimum samples needed to split a node.	(**2**,randuniform(0.0, 1.0))
Gradient Boosting	n_estimators	The number of boosting stages to perform.	(**100**, randuniform(50, 150))
	min_samples_leaf	The minimum samples needed to be at a leaf node.	(**1**, randuniform(1, 10))
Logistic Regression	penalty	To specify the norm used in the penalization.	(l1,**l2**)
	tol	The tolerance for the stopping criteria.	(**1e-4**, randuniform(0.0,0.1))
	C	The inverse of regularization strength.	(**1**, randint(1,500))
Adaboost	n_estimators	The maximum number of estimators at which the boosting is terminated.	(**50**, randuniform(50, 150))
Multinomial Naive Bayes	alpha	The additive smoothing parameter.	(**1**, randuniform(0, 1))
Decision Tree	splitter	Choosing strategy for splitting at each node.	(**best**, random)
	criterion	The function for measuring the split quality.	(**gini**, entropy)
	min_samples_split	The minimum samples needed to split a node.	(**2**, randuniform(0.0,1.0))
KNN	n_neighbors	The number of neighbors.	(**5**, randint(2, 25))
	weights	The weight function used in prediction.	(**uniform**, distance)
	p	Power parameter.	(**2**,randint(1,15))
	metric	The distance metric to use for the tree.	(**minkowski**, chebyshev)

performance (we use AUC) for the constructed reopened bug prediction pipeline. Shihab et al. used a Decision Tree learner without tuning the hyperparameters to build their reopened bug prediction pipeline. Several prior studies do not tune the hyperparameters of their model [11, 29, 31, 73, 105]. However, recent studies show that tuning the hyperparameters of a model is pivotal to ensure its optimal performance and interpretation [1, 27, 90, 91].

Step 5: Model performance evaluation. In this step, we measure the performance of various models on determining the likelihood of reopening a bug. We choose the AUC measure because the AUC measure is both threshold-independent and class imbalance insensitive. Several prior studies recommend the usage of AUC performance measure to evaluate the performance of the models in software analytics studies [30, 55, 73, 74, 91]. We find the performance of the reopened bug prediction pipeline on a given project to be acceptable if on the given project mean AUC value of the constructed reopened bug prediction pipeline across the 100 iterations is greater than 0.7 [3, 45, 66, 97]. Otherwise, we deem the performance of the reopened bug prediction pipeline as poor.

Step 6: Ranking important features. To generate the feature importance in predicting the reopened bugs, we use the permutation feature importance method [5]. We use the permutation feature importance method over the top node analysis used by Shihab et al. for the following reasons. First, Shihab et al. compute the feature importance ranks on re-balanced datasets. Second, the results of top node analysis can be biased since the Decision Tree favors categorical features [5]. To avoid these shortcomings, we use a permutation feature importance method. The permutation feature importance method works as follows. First, we compute the performance of reopened bug prediction pipeline. We then randomly permute the values of each feature at a time and compute the model's performance to note how much performance drop does permuting a feature encounters when compared to the model built on all the features whose values are

not permuted. A larger drop in performance due to the permutation of a feature's values signifies the high importance of that feature.

For computing the feature importance ranks, we use the dataset as it is to train the model (i.e., we do not use SMOTE in feature importance analysis). Tantithamthavorn et al. [90] find that using class re-balancing techniques when computing feature importance ranks results may lead to wrong features being considered as the important features. We compute the feature importance ranks only on the projects on which the best performing reopened bug prediction pipelines have an acceptable AUC as Lipton; Chen et al. [19, 57] argue that for a model to be interpreted, it needs to have an acceptable performance. After this step for each iteration of training data, feature importance scores are obtained for each feature. More details of the working of permutation feature importance method can be found in Altmann et al. [5]. After generating the feature importance score, we compute the features importance ranks using Scott-Knott ESD test [92]. These ranks determine the order of importance of features in predicting the reopened bugs. We finally note the most common top two most important features used by the best performing model across all the studied projects.

Results: Across the 47 studied projects, at best, we observe an acceptable AUC ($\geqslant$ 0.7) for 34% (16/47) of the projects. Table 3.3 shows the percentage of projects on which our constructed reopened bug prediction pipelines deliver an acceptable AUC. We observe that at most we achieve an acceptable AUC only on 34% of the studied projects, while on the remaining 66% projects a poor AUC ($<$ 0.7) for predicting reopened bugs is observed.

Reopened bug prediction pipeline constructed with Random Forest and Gradient Boosting learners yield an acceptable performance on more studied projects (AUC $\geqslant$ 0.7) than a pipeline with other learners. From Table 3.3, we observe that the Random Forest learner and Gradient Boosting learner yield an acceptable AUC on 34% of the studied projects. Whereas, the Decision Tree and KNN learners give an acceptable AUC on only 10.6% of the projects. Such a result argues that future studies should consider Random Forest and Gradient Boosting learners to construct reopened bug prediction pipelines.

Table 3.3: Percentage of projects with which the constructed reopened bug prediction pipeline with different learners yields an acceptable AUC

Model	**Acceptable projects (AUC$\geqslant$ 0.7) count (mean AUC)**	**Poor projects (i.e., AUC$<$ 0.7) count (mean AUC)**	**Percentage of the acceptable projects**
Random Forest	16 (0.78)	31 (0.63)	**34%**
Gradient Boosting	16 (0.77)	31 (0.63)	**34%**
Logistic Regression	14 (0.78)	33 (0.63)	29.8%
Adaboost	12 (0.79)	35 (0.63)	25.5%
Multinomial Naive Bayes	8 (0.75)	39 (0.61)	17%
Decision Tree	5 (0.82)	42 (0.59)	10.6%
KNN	5 (0.76)	42 (0.58)	10.6%

For the projects with an acceptable AUC, the *number of comments* is the most important feature. Such a result is in contrast with the findings by Shihab et al. who found that *comment text* is the most important feature in determining the likelihood of reopening a bug. We observe that using the best performing learner (i.e., Random Forest), the *number of comments* is the most important feature to predict reopened bugs in 31.3% of the projects with an acceptable AUC. We further find that the *comment text* which Shihab et al. found to be the most important feature in 66.7% (i.e., two out of

three) projects, is the most important feature in only 12.5% projects and second most important feature in 31.3% projects.

Summary of RQ1

Using our updated reopened bug prediction pipeline onto 47 JIRA projects, we find that only on 34% of the studied projects we are able to achieve an acceptable AUC ($\geqslant 0.7$) for our constructed reopened bug prediction pipeline. Furthermore, we observe that the Random Forest learner and Gradient Boosting learner give the best AUC in determining the likelihood of reopening a bug. Finally, we also observe that *number of comments* is the most important feature in determining the likelihood of reopening a bug.

3.3.2 RQ2: Why do bugs get reopened?

Motivation: The reopened bug prediction interpretation generates coarse insights in prior studies. For example, by interpreting the reopened bug prediction pipeline, we could determine that *number of comments* and *comment text* are the top two most important features in determining the likelihood of reopening a bug; however, such insights do not provide any actionable suggestions that can be used to either prevent bug reopening or identify the root cause of reopening bugs. Moreover, we observe that some developers pointed out the rationale to reopen the bug during reopening the bug. For example, Figure 3.4 shows a bug report[9] (ID: 4112) for the Apache Accumulo project. In this bug report, the bug was considered as non-reproducible initially and resolved but later a developer reopened the bug and at the same time commented about their rationale for reopening the bug. However, comments which are provided either

[9] https://issues.apache.org/jira/browse/ACCUMULO-4112

during bug reopening or after reopening a bug are not used in the reopened bug prediction pipeline in order to prevent data leak. This useful data (i.e., events during and post reopening bugs) can be analyzed using other approaches (such as a qualitative analysis) to generate insights into reopened bugs. Therefore, in this RQ, we investigate what are the rationale to reopen bugs. Moreover, since we observe in Section 3.3.1 that only 34% of the studied projects give an acceptable AUC for predicting reopened bugs, in this RQ, we also investigate if the reasons for reopening a bug differ between the projects on which the constructed reopened bug prediction pipelines have an acceptable and a poor performance. In order to better understand the rationale for reopening bugs, we leverage the whole history of bug reports to understand the characteristics of reopened bugs. We wish to gain insights into the challenges in building reopened bug prediction pipelines to better understand and manage reopened bugs.

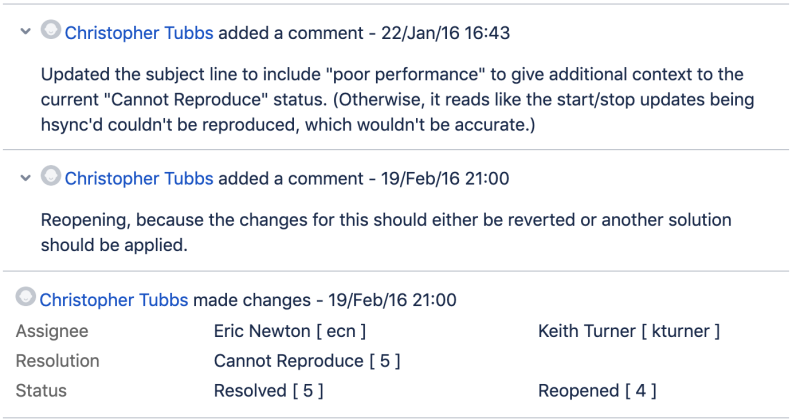

Figure 3.4: A bug report (ID: 4112) for the Apache Accumulo project, showing a comment depicting the reason for reopening the bug at the same time when the bug is reopened, hence this comment is not used in reopened bug prediction pipeline.

Approach: We use all the 9,993 reopened bugs from 47 projects in our analysis. A mixed

methods approach is used (including both the quantitative and qualitative study approach) in our study. We initially perform a quantitative analysis to identify the co-occurring changes at the time of reopening the bug. We then perform a qualitative analysis to identify the reasons why bugs get reopened.

Quantitative study: The aim of the quantitative study is to identify what are all the changes that occur in reopened bugs. For our quantitative study, we divide all the changes that occur in the bug reports into three categories (i.e., before reopening, during reopening, and after reopening). We consider all such events that occur before reopening a bug as the *before reopening* category and all such events that occur at the time of reopening a bug as the *during reopening* category. We consider all such events that occur after a bug is reopened as the *after reopening* category. We extract all the fields of changes during reopening a bug and identify the commonly occurring change patterns.

Qualitative study: For our qualitative study, we first randomly select reopened bugs, then conduct an open coding study [22] to understand the reasons that bugs get reopened.

There are 9,993 reopened bugs from the 47 projects on JIRA. Out of these bugs, we selected a random sample of 370 reopened bugs. Our selection reaches a statistically representative sample of all the reopened bugs associated with the 47 projects with a 95% confidence level and a 5% confidence interval [14, 118]. For our qualitative study, we examine all the associated comments with our manually studied bugs, i.e., comments before/during/after a bug is reopened. For example, in a bug report[10]

[10] https://issues.apache.org/jira/browse/FLEX-34059

(ID: 34059) for the Apache Flex project, a developer commented 40 minutes after the bug was reopened that: "*because the issue also happens with TextInputSkin (see ML piotr.zarzycki)*."

Three researchers (i.e., A1, A2, and A3 with me being one of them) conducted a manual study of tagging the reasons to reopen bugs. We aim to identify two labels, i.e., what is the reason to reopen a bug, and whether the rationale for reopening the bug occurred before, during, or after the bug is reopened? We discuss the steps in the open coding process below:

Step 1: Round-1 coding. We randomly select 50 bugs out of the 370 reopened bugs. We tag these bugs individually. Each researcher's tagging process is independent. We do not interfere in other researcher's tagging process.

Step 2: Discussion after Round-1 coding. After the first round, we obtained around 23 tags. These tags are specific reasons to reopen a bug. We then discuss to cluster these tags into categories of tags. After this meeting, we finalize the tag categories for the Round-1 coding.

Step 3: Revisit Round-1 coding. After the discussion, we re-code the same 50 bugs with the updated tag categories.

Step 4: Round-2 coding. The remaining 320 bugs are used for the Round-2 coding. A1, A2 and A3 are assigned with two-thirds of the remaining bugs (i.e., 214 bugs) each, such that each bug is tagged exactly by two researchers. We are allowed to add new tags if a bug can not be classified by the existing Round-1 tag categories. Round-2 coding is also done individually by each researcher without the influence of other researchers. It took

several hours for each one of the researchers to finish Round-2 coding.

Step 5: Discussion after Round-2 coding. After the Round-2 coding, we discuss the results and resolve any disagreement. After this meeting, we finalize the tags for our study.

Step 6: Revisiting Round-1 and Round-2 coding. After the discussion in the previous round, we finally revisit all the bugs of Round-1 and Round-2 coding to update the new tags obtained after Round-2 coding. This step took several hours for each researcher. This step is performed without the interference of other researchers so as to maintain the independence of decision while tagging bugs.

Step 7: Computing inter-researcher agreement. We use the tags to compute the inter-researcher agreement. The inter-researcher agreement is a measure to compute the agreement reached in tagging a bug. A bug reopening reason is considered as a match if both the researchers provide the same tag for a bug and a mismatch if both the researchers provide different tags for a bug. We compute the inter-researcher agreement using Krippendorff's α [8, 35]. The inter-researcher agreement for our study is 0.87 which concludes that our qualitative analysis is reliable as prior studies use $\alpha \geqslant 0.8$ as an indicator of reliable agreement [10, 56, 82, 99, 103].

Step 8: Dealing with disagreements. After coding all the bugs, we discuss the disagreements among the researchers for bugs with different tags. If a bug is tagged differently by two researchers, and they mutually do not agree to a common tag, then the tag of the bug is determined by the third researcher.

For understanding the rationale for reopening a bug, we use four predefined categories (i.e., before reopening, during reopening, after reopening, and no evidence found) as our tags. Each researcher tags the evidence for the reopened bugs that are assigned to him when labeling the reasons.

After completing the above qualitative study, we study the distribution of reasons for reopening bugs in projects with an acceptable AUC and projects with a poor AUC. To determine the projects that have an acceptable AUC and a poor AUC in the reopened bug prediction pipeline, we consider the findings from Section 3.3.1. For this study, we classify the projects as acceptable and poor based on the best performing learner (i.e., Random forest) identified in Section 3.3.1. We label all the 370 manually studied reopened bugs based on the project category as *reopened bug from acceptable project* and *reopened bug from poor project*. After labeling these 370 manually studied reopened bugs, we observe that 80 reopened bugs have a project category tag as *reopened bug from acceptable project* and the remaining 290 reopened bugs have a project category tag as *reopened bug from poor project*. To compare the distribution of reasons to reopen a bug in acceptable and poor projects, we need to use the same dataset as we used to train the reopened bug prediction pipeline. In our reopened bug prediction pipeline we only used the dataset before reopening the bug. Therefore, to compare the distributions of reasons to reopen bugs in acceptable and poor projects, we only use those bugs where the reason to reopen a bug was identified before reopening the bug. From our first qualitative study, we had obtained the evidence tags for reopened bugs with four predefined categories (i.e., before reopening, during reopening, after reopening, and no evidence found). Using these *evidence* tags and *project category* tags, we identified that 13 out of 80 reopened bugs with project category as *reopened*

bug from acceptable project have *evidence* tag as *before reopening*, and 44 out of 290 reopened bugs with project category as *reopened bug from acceptable project* have *evidence* tag as *before reopening*. Therefore initially, we consider only 13 reopened bugs in acceptable projects and 44 reopened bugs in poor projects. However, this sample of reopened bugs is too small for any meaningful conclusion from the study, therefore, we added more samples in each project category (i.e., *reopened bug from acceptable project* and *reopened bug from poor project*) of reopened bugs, until we obtained 50 reopened bugs with evidence tag as *before reopening* from acceptable projects and 50 reopened bugs with *evidence* tag as *before reopening* from poor projects.

Results: Reopened bugs have an average 2.4 number of other changes of the bug report during bug reopening. Apart from the obvious change of reverting the resolution (98.2%) and posting a comment (63%) at the time of reopening a bug, 12.2% of our studied bugs are reassigned during reopening. From a qualitative study, we observe that in 77.6% of the reopened bugs, the rationale for reopening is provided either during or after the reopening event to support why a bug was reopened. Using our quantitative study, we identify all the field changes that occur in reopened bug reports. Table 3.4 shows the distribution of field changes in the reopened bugs in different time periods (i.e., before reopening, during reopening, and after reopening). While analyzing the reopened bugs, we observe that 19.2% of the reopened bugs are reassigned at least once after being reopened. Yadav et al. [111] observe that reassigning bugs during bug fixing is not always harmful since bug reassignment can involve knowledge transfer among developers. In our study, we observe that in 2.3% (226 out of 9,993) of the reopened bugs, developers change the assignee of the bug and

resolve/close the bug immediately, indicating that the bug was reopened solely for the purpose of updating the assignee field of the bug report; however, the bug was already fixed. For example, in a bug report[11] (ID: 741) for the Apache Airvata project, the bug assignee was changed while reopening the bug and the bug was immediately closed. In 9.9% (990 out of 9,993) of the reopened bugs, the developers change the assignee and do not immediately close/resolve the bug, suggesting that the bug was not fixed and re-assigned to another developer during reopening. For example, in a bug report[12] (ID: 4640) for the Apache Accumulo project, the bug was re-assigned at the time of reopening the bug; however, the new assignee worked on the bug later and eventually resolved the bug. Prior studies observe that with an increasing number of bug reassignments, time to fix the bug also increases [33, 79]. The bug resolution process (i.e., during bug reopening and after bug reopening) is rich and contains many insights about bug reopening activities; however, no prior study has used this dataset to generate insights into bug reopening.

Table 3.4: The distribution of major field changes that occur in reopened bugs grouped by before, during, and after reopening a bug.

Change field	Before	During	After
Resolution	9,632 (96.4%)	9,810 (98.2%)	8,227 (82.3%)
Comment	9,371 (93.8%)	6,301 (63.1%)	8,666 (86.7%)
Assignee	4,956 (49.6%)	1,216 (12.2%)	1,916 (19.2%)
Fix version	5,053 (50.6%)	1,072 (10.7%)	4,962 (49.7%)
Attachment	3,750 (37.5%)	211 (2.1%)	2,104 (21.1%)
Link	2,144 (21.5%)	199 (2%)	1,796 (18%)

71.1% (i.e., 263 out of 370) of the manually studied reopened bugs, the rationale

[11] https://issues.apache.org/jira/browse/AIRAVATA-741
[12] https://issues.apache.org/jira/browse/ACCUMULO-4640

for reopening is provided at the time of reopening the bug. Table 3.5 shows the rationale with their bug count and proportion. For example, in a bug report[13] (ID: 2197) for the Apache OFBiz project, the developer posted a comment about reopening the bug at the time of reopening the bug: "*Reopened issue to re-examine the latest patch of CSS changes to the bluelight theme.*" In another example, a bug report[14] (ID: 11429) for the Apache Ignite project was reopened to update the resolution from *fixed* to *duplicate*, which occurred at the same time as reopening the bug. In 6.5% (i.e., 24 out of 370) of the studied reopened bugs, the rationale is provided after the bug is reopened. For example, in a bug report[15] (ID: 3598) for the Apache NetBeans project, after reopening the bug the developer discussed that the patch did not work.

Although we observed in Section 3.3.1, that *comment text* is among the most important features in determining the likelihood of reopening a bug, many rationales in the history of bug reports can't be leveraged in determining the likelihood of reopening a bug as they occur mainly at the time of bug reopening or even after a bug was reopened. However, we encourage future research to leverage this set of data (i.e., during and after bug reopening) to alleviate the reopened bug management.

We identify four categories of reasons for reopening bugs, including technical (i.e., patch/integration issues), documentation (bookkeeping), human, and no stated reason. Table 3.5 shows the reason categories together with their number/proportion, explanation, and examples. We discuss the reasons for reopening bugs below:

[13] https://issues.apache.org/jira/browse/OFBIZ-2197
[14] https://issues.apache.org/jira/browse/IGNITE-11429
[15] https://issues.apache.org/jira/browse/NETBEANS-3598
[16] https://issues.apache.org/jira/browse/HBASE-14689
[17] https://issues.apache.org/jira/browse/AMBARI-14383
[18] https://issues.apache.org/jira/browse/HAWQ-227
[19] https://issues.apache.org/jira/browse/HADOOP-10589
[20] https://issues.apache.org/jira/browse/STORM-190

Table 3.5: Different reasons for reopening the bugs based on various rationale categories (i.e., before bug reopening, during bug reopening and after bug reopening

Category	Reason	Explanation	Example	# projects	Overall count (%)	Before reopening count (%)	During reopening count (%)	After reopening count (%)
Technical	Patch	Bug is not fixed by the patch.	Reopening, since it seems this issue is causing problems with high load. Will revert and test.[16]	37	202 (54.6%)	50 (13.5%)	142 (38.4%)	10 (2.7%)
	Integration	Bug is reopened because of an issue during the patch integration process.	This has been reverted, it breaks unit tests.[17]					
Documentation	Update	Bug is reopened to update a field.	reopen to update version[18]	31	90 (24.3%)	3 (0.8%)	78 (21.1%)	9 (2.4%)
Human	Incorrect assessment	Bug is reopened due to incorrect assessment.	not a duplicate. Same stack trace, but root cause is different[19]	28	52 (14.1%)	4 (1.1%)	43 (11.6%)	5 (1.4%)
No reason	No reason	No reason was found in the bug report for reopening.	Not A Problem[20]	15	26 (7%)	-	-	-
Total				47	370 (100%)	57 (15.4%)	263 (71.1%)	24 (6.5%)

• **Inline with common intuition, the majority (i.e., 54.6%) of our studied bugs are reopened due to technical issues during the bug fixing processes, that is, the failure of code patch and patch integration.** More specifically, 41.6% (i.e., 154 out of 370) of the reopened bugs are due to patch issues. Some of the major patch issues include runtime errors, regression, code improvements, and code merge issues. For example, a bug report[21] (ID: 5206) for the Apache Flink project is reopened due to runtime exception. Patch issues can deprecate the quality of the system and hence need to be fixed before resolving the bug. However, the above bug was not carefully tested before it was marked as fixed. During bug reopening, a developer pointed out that there

[21] https://issues.apache.org/jira/browse/FLINK-5206

is a runtime exception issue in the system and it took around 1.3 years to fix the reopened bug. In addition, 13% (i.e., 48 out of 370) of our studied reopened bugs are due to integration failure. Integration issues include build failures, tests failure, flaky tests[22], and unit test failures. For example, in a bug report[23] (ID: 346) for the Apache Daffodil project, test cases failure leads to the bug reopening. In this bug, a developer had already warned to test the fix before closing the issue; however, after closing the bug, a developer found that test cases are failing leading to reopening the bug. It took three days extra to resolve the issue, from the time the test cases failure issue was first observed. Stolberg [89] suggested certain code integration testing techniques such as running all unit tests with every build and developing unit tests for all new code that allow fixing bugs at a cheaper cost. We suggest the developer community to follow such code integration testing techniques during the bug fixing process in order to avoid reopening a bug later due to integration issues.

• **In 24.3% (i.e., 90 out of 370) of the studied reopened bugs, bugs are reopened due to bookkeeping issues (i.e., updating a field in the bug report).** Bug reports have various bug fields, such as *version*, *assignee*, and *resolution*. In order for developers to update these bug fields, the bug needs to be reopened. We observe that 10.5% (i.e., 39 out of 370) bugs are reopened to update the fix version. For example, a bug report[24] (ID: 480) for the Apache HAWQ project was reopened just to update its associated version. Moreover, 7% (i.e., 26 out of 370) bugs are reopened to update the resolution. For example, a bug report[25] (ID: 1999) for the Apache Felix project was reopened and the developer who reopened the bug posted a comment to point out that it was reopened

[22] https://whatis.techtarget.com/definition/flaky-test
[23] https://issues.apache.org/jira/browse/DAFFODIL-346
[24] https://issues.apache.org/jira/browse/HAWQ-480
[25] https://issues.apache.org/jira/browse/FELIX-1999

to fix the resolution. 1.4% (i.e., 5 out of 370) bugs are reopened to update the assignee. For example, in a bug report[26] (ID: 3598) for the Apache Spark project, the bug was reopened to change its assignee. Bugs that are reopened to update the meta information of a fixed bug can use other mechanisms to avoid any confusion with a reopened bug due to technical issues. So far, technical issues and documentation issues are not differentiated by prior studies in determining the likelihood of reopening a bug. New bug features can be used to better identify reopened bugs due to technical issues instead of documentation issues.

• **In 14.1% (i.e., 52 out of 370) of our studied reopened bugs, bugs are reopened due to an incorrect initial assessment.** These bugs include cases where a bug was wrongly assessed initially, leading to the reopening of the bug at a later stage. We observe that 3.8% (i.e., 14 out of 370) of our studied bugs were reopened since they were wrongly classified as *non-reproducible*. In this case, if a developer is not able to reproduce the bug, then he stops working on the bug and resolves the bug without fixing it by considering the bug as *non-reproducible*. For example, in a bug report[27] (ID: 2570) for the Apache Thrift project, a developer initially closed the bug by mentioning "*can't reproduce ... feel free to re-open if new information becomes available*", but after almost a month, another developer reopened the bug and fixed the bug. To handle such issues, the reporter of the bug should provide detailed reproduction steps for assignees to work on the bug properly. 2.7% (i.e., 10 out of 370) of our studied bugs were wrongly considered as a duplicate. For example, a bug[28] (ID: 17024) for the Apache

[26] https://issues.apache.org/jira/browse/SPARK-3598
[27] https://issues.apache.org/jira/browse/THRIFT-2570
[28] https://issues.apache.org/jira/browse/SPARK-17024

Spark project is considered as a duplicate initially; however, after 8 days, another developer reopened the bug and started working on fixing the bug. Prior work [40] also studied duplicate bug reports and found that in some projects almost a quarter of all bugs are duplicate.

• **In 7% (i.e., 26 out of 370) of our studied reopened bugs, we were not able to identify the reason for reopening the bug.** In these bugs, developers do not provide any rationale in the bug to identify the reason for reopening the bug. For example, in a bug report[29] (ID: 1152) for the Apache HAWQ project, no comment is provided at the time of reopening the bug.

In projects with an acceptable AUC, almost 94% and a negligible 4% of the bugs where the reason to reopen the bug was identified before reopening the bug are reopened due to patch issues and integration issues respectively; however, in projects with a poor AUC, around 66% and 22% of the bugs where the reason to reopen the bug was identified before reopening the bug are reopened due to patch issues and integration issues respectively. Table 3.6 shows the distribution of reason categories identified before reopening the bug together with their number/proportion for both acceptable as well as poor projects. In the projects where we get an acceptable AUC (i.e., acceptable projects), we manually classify the bugs with reasons before the reopening event. We observe that in 50 out of 330 randomly studied reopened bugs from acceptable projects, the reason to reopen bugs was identified before bug reopening. Out of these 50 bugs, there are 49 (98%) bugs where the reason to reopen a bug was identified before bug reopening are reopened due to technical issues; however, no bug is identified where the reason identified before bug reopening is due to documentation

[29] https://issues.apache.org/jira/browse/HAWQ-1152

issues. For example, in a bug report[30] (ID: 2626) for the Apache Cassandra project, the developer comments before reopening the bug that *"Thanks for your quick response. I'm not sure but I think that the fix doesn't work. Types of pendingFlush, sstables and compacting objects are java.util.Collections.UnmodifiableSet. Those are not instances of org.apache.commons.collections.set.UnmodifiableSet."* stating that the bug was reopened due to technical reasons. In projects with an acceptable AUC, a technical reopening reason can be identified before reopening events, suggesting that new features that can characterize technical issues can be used in a reopened bug prediction pipeline. Moreover, we observe that in 50 out of 313 randomly studied reopened bugs from poor projects reason to reopen a bug was identified before bug reopening. Out of these 50 bugs, 11 (22%) are reopened due to integration issues, this is higher than the amount of bugs reopened due to integration issues (i.e., 4%) identified before bug reopening in projects with an acceptable performance. Apart from bugs reopening in projects with poor performance due to technical issues, 3 (6%) bugs where the reason to reopen a bug was identified before bug reopening are reopened due to documentation issues. For example, in a bug report[31] (ID: 15519) for the Apache Spark project, a developer comments *"I misclicked and resolved as Fixed instead of as Duplicate. Feel free to edit."* Since bug reopening due to documentation issues is a type of trivial bookkeeping activity, we encourage future studies to remove the bug reopens due to documentation issues from their study of determining the likelihood of reopening a bug.

From the study using the before reopening bug dataset, we observe that 98% and 88% of bugs in projects with an acceptable and a poor performance respectively are

[30] https://issues.apache.org/jira/browse/CASSANDRA-2626
[31] https://issues.apache.org/jira/browse/SPARK-15519

reopened due to technical issues. All these reasons are identified by developer comments, suggesting that *comment text* is an important feature in the study of reopened bugs, which supports our finding in RQ1. From the study using during/after bug reopening dataset and before bug reopening dataset, we observe that in 77.6% of the bugs the reason to reopen a bug is identified during/after reopening the bug, this is 5 times more than bug reopening reasons identified before (i.e., 15.4%) bug reopening, indicating that 5 times more data can be leveraged while considering during/after bug reopening dataset in future bug reopening studies. Moreover, it also signifies that useful discussions (i.e., comments about bug reopening reasons) occur more frequently during/after bug reopening than before bug reopening.

Table 3.6: Different reasons for reopening the bugs identified before bug reopening in various project categories (i.e., acceptable, and poor)

Category	**Reason**	**Evidence time**	**Acceptable performance projects**	**# Extra reopened bugs needed in acceptable projects**	**Poor performance projects**	**# Extra reopened bugs needed in poor projects**
Technical	Patch	before	47(94%)	250	33 (66%)	23
	Integration	before	2 (4%)	250	11 (22%)	23
Documentation	Update	before	0 (0%)	250	3 (6%)	23
Human	Incorrect assessment	before	1 (2%)	250	3 (6%)	23
No reason	No stated reason	before	-	250	-	23

Summary of RQ2

In 77.6% of the reopened bugs, the reason to reopen a bug can be identified either during or after a bug is reopened; however, this rich dataset (i.e., during and after bug reopening) is never studied specifically. 54.6% of our manually studied reopened bugs are due to technical issues (i.e., the initial patching/integration process fails). 24.3% of our manually studied reopened bugs are due to documentation issues (i.e., updating a field of the bug). 14.1% of our manually studied reopened bugs are due to human related issues (i.e., incorrect assessment). In projects with an acceptable AUC, 94%, and 4% of the bugs where the reason to reopen a bug is identified before bug reopening are due to patch issues and integration issues respectively, while in projects with a poor AUC, apart from patch issues (i.e., 66%), bugs are reopened due to integration issues (i.e., 22%).

3.4 Discussion

Table 3.7 shows the findings of our study and their implications.

3.4.1 Implications for developers

Developers should use code integration testing techniques such as running all unit tests with builds before resolving bugs in order to avoid bug reopening later due to integration issues. We observe in RQ2 (Section 3.3.2) that 13% (i.e., 48 out of 370) of the manually studied reopened bugs are due to the integration issues. Moreover, we also observe in RQ2 that 22% of the bugs in projects with poor AUC are reopened due

Table 3.7: Our major findings and their implications.

Audience	**Findings**	**Implications**
Developers	13% (i.e., 48 out of 370) of the manually studied bugs are reopened due to the integration issues and 22% (11 out of 50) of the bugs in projects with poor AUC are reopened due to integration issues.	Developers should use code integration testing techniques such as running all unit tests with builds before resolving bugs in order to avoid bug reopening later due to integration issues.
Researchers	63.1% and 86.7% bugs have comments during and after bug reopening respectively. In 77.6% bugs the reason to reopen a bug is identified either during or after reopening the bug.	Researchers should leverage data from during/after bug reopening to determine challenges in reopened bug management.
	24.3% of the bugs are reopened to update the fields of the bug report.	Researchers should drop the subdataset containing bug reports reopened due to documentation (i.e., bookkeeping) issues.
	Prior studies on the prediction of reopened bugs are based on dataset with data leak issues.	Researchers should consider only those events that occur before a bug is reopened to predict reopened bugs.

to integration issues identified before bug reopening. These issues include build failure, tests failure, unit tests failure, flaky tests, and tests timed out. Some of these issues can be handled during the bug fixing process, such as by executing unit tests locally and only committing the patch when the unit tests pass. Developers can also use automated tools for verifying the test accuracy. Stolberg [89] suggested certain code integration testing techniques such as running all unit tests with every build and developing unit tests for all new code that allow fixing bugs at a cheaper cost. Developers can use such techniques to resolve the bugs in order to avoid bugs getting reopened due to integration issues. Other issues, particularly the build success can also be tested after

committing the patch. However, developers may be careless in performing such tests, or simply skip them to directly resolve the bugs. Additionally, we observe that there is a varying trend among development teams in waiting for the build to succeed and then resolving bugs. For example, in a bug report[32] (ID: 4531) for the Apache Thrift project, the developer waits for the build to succeed then resolves the bug. Whereas, in a bug report[33] (ID: 15977) for the Apache HBase project, the developer resolves the bug immediately after committing the patch without waiting for the build to succeed, later the build fails leading to reopening the bug. To avoid build failures in the master branch, the developer can use a feature branch to test their patch before merging the patch to the master branch. For example, in a bug report[34] (ID: 1716) for the Apache Geode project, the developer tests the patch in feature branch, and later the bug is resolved. A better mechanism to control the quality of test/build processes in bug fixing is needed so that developers can avoid reopening bugs later on.

3.4.2 Implications for the researchers:

Researchers should leverage data from during/after bug reopening to better understand the challenges surrounding reopened bug management. We observe in RQ2 (Section 3.3.2) that, 63.1% of bugs have comments during reopening, and 86.7% of bugs have comments after bug reopening. These during and after reopening comments which are not used in reopened bug prediction studies can be used to understand developer discussions of reopened bugs in order to determine the cause of bug

[32] https://issues.apache.org/jira/browse/THRIFT-4531
[33] https://issues.apache.org/jira/browse/HBASE-15977
[34] https://issues.apache.org/jira/browse/GEODE-1716

reopening and its remedy. For example, in a bug report[35] (ID: 5011) for the Apache Zeppelin project, the bug is reopened and later the developer comments *"The build worked after I changed -Pspark-3.0 to -Pspark-3.0.0. At the end I learnt that it had ignored -Pspark-3.0.0 as an invalid option."* indicating the resolution of the reopened bug. In addition, we observe that 77.6% of our manually studied bugs have reason to reopen the bug during or after reopening the bug. A lot of this untapped data can be used to understand the challenges faced by developers in reopened bugs.

Researchers should drop the sub-dataset containing bug reports reopened due to documentation (i.e., bookkeeping) issues. We observe in RQ2 that one of the reasons to reopen bugs is documentation issues. For example, in a bug report[36] (ID: 889) for the Apache HAWQ project, the developer reopened the bug to update the fix version. Some bugs are reopened in order to change a trivial bug report field, whereas other bugs are reopened in order to rework on the failed attempt to fix a bug. Bugs that are reopened due to documentation issues must be exempt from the reopened bug prediction pipeline in order for the prediction pipeline to predict more relevant bug reopening activities.

Researchers should consider only those events that occur before a bug is reopened to predict reopened bugs. We observe that in prior studies [84, 105], one out of three studied projects (i.e., Apache project) has bugs with the last status as *reopened* to determine the likelihood of reopening a bug. This is a data leak issue, and the bugs with a data leak issue should not be considered in the prediction pipeline. We suggest the research community carefully use data in prediction pipelines, especially in the case when events occur sequentially.

[35] https://issues.apache.org/jira/browse/ZEPPELIN-5011
[36] https://issues.apache.org/jira/browse/HAWQ-889

3.5 Threats

3.5.1 External Validity

Our selection criteria involve the top 50 projects in JIRA based on the committers count. The committers count criteria may not be the only metric to indicate the popularity of projects. For example, some committers may not be active, leading to projects with highly active but less number of committers. Future work can leverage other criteria that can be used to select projects. The selection of our studied dataset has an impact on the model performance and the feature importance analysis findings in our study.

3.5.2 Internal Validity

We study the reasons why bugs get reopened in RQ2 (Section 3.3.2). We select a sample of bugs for our qualitative analysis by examining 370 bug reports from 9,993 reopened bugs. Other bugs that are not part of our studied sample may contain reasons other than what we identified in our analysis. However, we minimize the sample bias by selecting a statistical representative sample with a 95% confidence level and a 5% confidence interval. In our study, we do not claim to find all the reasons to reopen a bug. To alleviate this threat, future research is encouraged to study a larger sample of reopened bugs.

3.5.3 Construct Validity

We considered 7 learners (i.e., Random Forest, Gradient Boosting, Logistic Regression, Adaboost, Multinomial Naive Bayes, Decision Tree, and KNN) to predict the reopened

bugs. However, there are other learners that are not covered in determining the likelihood of reopening a bug. The selection of 7 learners for our study is motivated by the study conducted by Agrawal et al. [1]. Agrawal et al. used five learners (i.e., Random Forest, Logistic Regression, Multinomial Naive Bayes, Decision Tree, and KNN) for their study, we also consider two more learners (i.e., Gradient Boosting, and Adaboost) in our study as they are extensively used in prior prediction studies [29, 90, 91].

We use the AUC value of 0.7 as a threshold to determine projects with an acceptable performance in determining the likelihood of reopening a bug. Changing the threshold value can change the percentage of projects with an acceptable performance, and impact the relative performance of different learners. However, we observe that many prior studies consider 0.7 as an AUC threshold to be acceptable [3, 45, 66, 97].

3.6 Chapter Summary

A reopened bug can take considerably more time to fix, and developers can make a lot of effort in fixing reopened bugs. In this chapter, we revisit reopened bug prediction on a large dataset consisting of 47 projects tracked by JIRA and using an updated pipeline for reopened bug prediction. We share the findings of our study: 1) Prior studies on reopened bug prediction are based only on three projects (i.e., Apache, Eclipse, and OpenOffice), and out of these projects, the Apache project has a data leak issue. Hence, prior studies on predicting reopened bugs are effectively based only on two projects. Moreover, these studies used old techniques for constructing a reopened bug prediction pipeline, indicating that predicting a reopened bug is not a solved problem. 2) Using our updated reopened bug prediction pipeline, we observe that only 34% of our studied 47 projects have an acceptable performance (i.e., AUC $\geqslant$ 0.7) in determining

the likelihood of reopening a bug. 3) For the projects where the reopened bug prediction does not perform well, we propose researchers create new features representing technical (i.e., specifically patch related) and documentation (i.e., bookkeeping) issues and leverage them to predict reopened bugs.

Based on the findings in our study, we suggest the developer community follow practices involving carefully examining the proposed patch and testing the build success before resolving a bug. We also encourage future research to explore this new venue of reopened bugs, e.g., by proposing new explanatory models to understand reopened bugs from leveraging the whole history of bug report and exploring insights into fixing reopened bugs more efficiently.

CHAPTER 4

Studying post release reopened bugs in open source software projects

Reopened bugs can be categorized into two categories based on release, that is, pre release reopened bugs and post-release reopened bugs. Post-release bugs are more impactful than pre release bugs as they affect a large number of audience (i.e., clients) as compared to only development team in case of pre release bugs. Rwemalika et al. observe that post-release bugs need at least two times bigger chunks of changes as compared to pre release bugs. However, no prior study on reopened bugs specifically considered post-release reopened bugs for their analysis. Therefore, to further study in this area, we perform analysis on reopened bugs by segregating them based on release. We base our study on 23,350 reopened bugs from 462 projects hosted on JIRA. We first

analyze the characteristics of pre release and post-release reopened bugs. Furthermore, we build prediction pipelines on post-release reopened bugs to predict which of the post-release reopened bugs will get resolved rapidly and which of post-release reopened bugs might get resolved slowly and thereby need more attention. We also interpret these models and provide feature importance ranks for these models. We find that: 1) Post-release reopened bugs are 2.6 times more abundant than pre release reopened bugs. 2) For pre release reopened bugs the median rework time is twice as much as post-release reopened bugs. 3) Pre-release reopened bugs take 57.7%, 84.6%, and 87.6% less rework time than post-release reopened bugs when the bug is reopened due to patch, integration, and incorrect assessment issues respectively. 4) We are able to predict whether a post-release reopened bug will get resolved rapidly (i.e., less than 3.3 minutes) or slowly (i.e., more than 4,538 hours) with an acceptable AUC of 0.78. 5) Features related to developer comments (i.e., *comments text*, and *comments size*) and bug description (i.e., *description text*) are among the top three most important features to predict if a post-release reopened bug will get resolved rapidly or slowly.

We observe that the required time to resolve pre and post-release reopened bugs measured is dramatically different. Studying reopened bugs as a whole leads to biased implications. Therefore, we encourage researchers to consider pre-release and post-release reopened bugs separately in their analysis.

4.1 Introduction

Software maintenance activities are estimated to take almost 85%-90% of the software life cycle cost [26]. Software developers tag the version of the release of the software system using the *affected version* field in JIRA bug reports. A bug can be pre-release or

post-release bug depending on when the software system's affected version tagged to the bug is released. If the affected version is tagged in a bug report before the release of the software system, the bug is considered as a pre-release bug, if the affected version is tagged in a bug report after the release of the software system, the bug is considered as a post-release bug.

Reopening a bug across the software development life cycle can potentially affect the reputation of software in different ways. For instance, in an example case A, a bug is opened before the release of a software and it gets closed before the software is released. The bug is then reopened after the software is released. In this case, a user can encounter this bug once in the field (after the release). In an example case B, if a bug is opened after the release of a software and later it gets closed and reopened. A user encounters this bug twice in the field. In case A, the reputation of the software is affected once due to the surfacing of the reopened bug; whereas in case B, the reputation of the software is affected further due to the repeated surfacing of the reopened bug.

Resolving a post-release bug is critical for development teams since post-release bugs affect a very large audience (i.e., clients). Post-release bugs are those bugs that escaped to production [78]. It shows that these bugs bypassed the basic testing of the software system. Rwemalika et al. [78] observe that post-release bugs need at least two times bigger chunks of changes as compared to pre-release bugs and 66.7% of the post-release bugs require changes in configuration files as compared to only 46% configuration file changes in case of pre-release bugs. Therefore, it is important for the research community to focus on post-release bugs.

Generally, bugs are worked upon and resolved finally, however, sometimes bugs need to be reopened. Reopened bugs are considered as a disappointment to the end

users in software development since it was advertised that a particular bug was fixed, but in reality, it is otherwise and it needs to be reopened and reworked [64]. A pre-release bug that is also reopened at any point of time is considered as a pre-release reopened bug and a post-release bug that is also reopened at any point of time is considered as a post-release reopened bug. Figure 4.1 shows a post-release reopened bug report[1] (id: 2325) for the Apache Zookeeper project. In this bug, affected version 3.4.6 is tagged to the bug at the time of bug creation (i.e., 18-Nov-2015). However, the affected version was released earlier on 10-Mar-2014.

Prior studies focused on reopened bugs as a whole. Shihab et al. [83] conducted an initial study to predict if a bug will get reopened in the Eclipse project. Shihab et al.; Xia et al. [84, 105, 107] also built models on predicting if a bug will get reopened. Mi and Keung [64] analyzed time spent in various statuses of reopened bugs from previous statuses (i.e., time to assign a bug, time to resolve a bug, time to reopen a bug). However, traditional defect prediction studies consider pre-release and post-release bugs separately in their analysis [49, 50, 80]. They consider post-release bugs to have different characteristics than pre-release bugs and consider post-release bugs as more important to be studied. By the same logic, analyzing reopened bugs as a whole in prediction studies ignores the importance of post-release bugs and can introduce bias in model performances and model interpretation w.r.t prediction pipeline based studies for more critical post-release reopened bugs. As per our knowledge, no prior study focused on analyzing the reopened bugs based on their release. Therefore, it becomes important to segregate the reopened bugs study based on release and independently study characteristics of pre-release reopened bugs and post-release reopened bugs.

All prior prediction studies on reopened bugs focused on predicting if a bug will

[1] https://issues.apache.org/jira/browse/ZOOKEEPER-2325

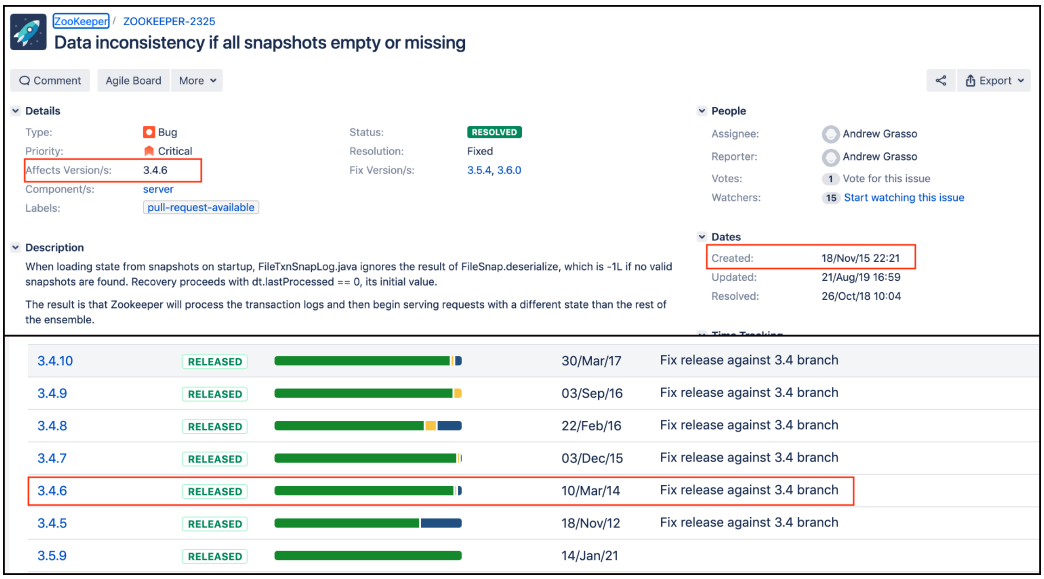

Figure 4.1: A post-release reopened bug report (id: 2325) for the Apache Zookeeper project.

get reopened. However, one of the goals of every development team is to provide a fix to the reopened bug and resolve a reopened bug rapidly. Predicting if a post-release reopened will get resolved rapidly/slowly can help software developers in taking decisions about allocating more resources for resolving slow post-release reopened bugs. Hence, it is also important to understand if we can predict which of the post-release reopened bugs will get resolved rapidly and which of post-release reopened bugs might get resolved slowly and thereby need more attention. This motivated us to do a study to find characteristics of pre-release and post-release reopened bugs and build a prediction pipeline to predict if a post-release reopened bug will get resolved rapidly/slowly. For our study, we consider all 23,350 reopened bugs from 462 projects in JIRA. We study the following RQs:

RQ1: What are the characteristics of pre-release reopened bugs and post-release reopened bugs?

Motivation: In software systems pre-release bugs and post-release bugs are handled differently. Taking a cue from the software developers, research also focused on handling pre-release and post-release bugs separately [49, 50, 80]. However, no such distinction is made in prior studies on reopened bugs [83, 84, 105, 106]. Therefore, we determine whether it is fine to consider reopened bugs as a whole in a reopened bugs study or whether segregating reopened bugs into pre-release reopened bugs and post-release reopened bugs is necessary. Most prior studies on reopened bugs focused on predicting if a bug will get reopened [83, 84, 105, 106]. However, to determine how long a reopened bug takes to finally get resolved will provide useful insights to development teams about bug resource allocation. Moreover, bugs can get reopened due to various reasons. However, no prior study focused on the analysis of distribution of these reasons to reopen bugs and the rework time of these reasons to reopened bugs on release based reopened bugs (pre-release reopened bugs and post-release reopened bugs). Therefore, in this RQ, we determine the characteristics of pre-release reopened bugs and post-release reopened bugs based on rework time, and analysis of rework time in the distribution of reasons to reopen bugs.

Results: 44.2% (i.e., 13,705 out of 31,013) reopened bugs are post-release reopened bugs, whereas only 16.9% (i.e., 5,249 out of 31,013) reopened bugs are pre-release reopened bugs. For the pre-release reopened bugs, the median time taken for rework (i.e., from first reopened to final resolved) is 388.4 hours, whereas for the post-release reopened bugs, the median time taken for rework is 189.1 hours. The

median time difference between rework and initial (i.e., from bug creation to initial bug resolved) is 31.9 hours in pre-release reopened bugs as compared to 0.3 hours in post-release reopened bugs. Through a qualitative study of 100 pre and post release reopened bugs each, we observe that in post-release reopened bugs, technical-patch, technical-integration, and incorrect assessment related issues take more than 271.1 hours, 1839.3 hours, and 4285 hours respectively to rework as compared to pre-release reopened bugs. Bugs reopened due to documentation issues, take less than eight minutes median time to rework in both pre and post-release reopened bugs.

RQ2: Can models predict if a post-release reopened bug will get resolved rapidly/slowly with an acceptable performance?

Motivation: Rwemalika et al. [78] observe that post-release bugs require at least two times bigger chunks of changes as compared to pre-release bugs. Prior studies [23, 50, 52, 113] considered only post-release bugs for their analysis. Post-release bugs garner more developer attention than pre-release bugs that's why research has focused on studying various aspects of post-release bugs such as their fixing time [49, 50, 80]. In the context of reopened bugs, we observe in RQ1 that pre-release reopened bugs and post-release reopened bugs are completely different in a plethora of ways. For example, the median difference between time spent in rework and time spent in initial resolving of a bug is only 0.3 hours in case of post-release reopened bugs as compared to 31.9 hours in case of pre-release reopened bugs. However, prediction studies on reopened bugs studies take pre-release and post-release reopened bugs together [83, 84, 105, 106]. However, this is in contrast

with the traditional defect prediction studies where researchers don't consider pre-release bugs and post-release bugs together [49, 50, 80]. Prior studies of software bugs are focused on predicting the lifetime of a bug [31, 51, 59, 72, 76]. However, as per our knowledge, no prior study focused on predicting the resolving time of reopened bugs. If we can predict well whether a post-release reopened bug will get fixed rapidly or slowly, we can notify developers about slow resolving post-release reopened bugs. The developers can then allocate more effort and resources towards resolving post-release reopened bugs with slow resolving time and allocate more resources towards those reopened bugs. Moreover, we also interpret the prediction pipeline to understand what characteristics one can look for to identify if a post-release reopened bug will get fixed fast or slow.

Results: At best we are able to predict if a post-release reopened bug will get resolved rapidly/slowly with an AUC of 0.78. Out of 7 studied learners (i.e., classifiers), 5 learners (i.e., Random Forest, Gradient Boosting, Logistic Regression, Adaboost, and Multinomial Naive Bayes) yield an acceptable AUC ($\geqslant$0.7) to predict if a post-release reopened bug will get resolved rapidly/slowly. *Comments size*, *comments text*, and *description text* are among the top three important features in 100%, 80%, and 60% of learners with an acceptable AUC ($\geqslant$0.7) respectively.

We observe that the required time to resolve pre-release and post-release reopened bugs is different. Prior studies on reopened bugs did not segregate pre-release and post-release reopened bugs for their analysis. Studying reopened bugs as a whole leads

to biased implications. Hence, we encourage the research community to consider pre-release and post-release reopened bugs separately in their analysis. Moreover, post-release reopened bugs are critical to resolve rapidly as they affect a very large audience (i.e., clients). Hence, we recommend future researchers in this field analyze developer comments in order to find reasons for the slow resolution of post-release reopened bugs.

The rest of the chapter is organized as follows. Section 4.2 showcases the data collection and classification processes. Section 4.3 discusses our case study. Section 4.4 exhibits a discussion about our findings. Section 4.5 showcases the threats to the validity of our study. Finally, Section 4.6 provides the conclusion of the chapter.

4.2 Data Collection

We discuss our data collection, data preprocessing, and data classification approach in this step. We collect reopened bugs dataset using JIRA bug REST API. In Section 4.2.1, we show the approach to collect the JIRA reopened bug dataset. We also collect version release dataset using the JIRA version REST API. In Section 4.2.2 we present our approach to collect the JIRA version release dataset. The JIRA reopened bug dataset contains text features. In Section 4.2.3 we show the preprocessing steps to clean these text features. In Section 4.2.4 we classify reopened bugs into pre-release reopened bugs and post-release reopened bugs.

4.2.1 Collecting JIRA reopened bug dataset

We study reopened bug projects hosted on JIRA. We utilize the JIRA bug REST API[2] to extract JIRA reopened bugs dataset. We collected all bugs that were ever reopened on JIRA. We collected 23,350 reopened bugs from 462 projects. We use all the features except *Number of files in the fix* from our previous study on the prediction of reopened bugs (Tagra et al.). Feature *Number of files in the fix* is computed using Github project repositories. Since we use all the 462 projects in our study, it is very difficult to manually find the Github repository for each project. Hence, we ignore this feature in our study of predicting if a release based reopened bug will get resolved rapidly/slowly. Table 4.1 shows the 19 features that we collect for our study.

4.2.2 Collecting JIRA version release dataset[3]

We use the JIRA version REST API to extract JIRA version release dataset[4]. This REST API requires project name as input to retrieve the release versions and dates. From Section 4.2.1 we collected the project names of all 462 projects. We then used these project names as an input to the JIRA version REST API and generate all release versions for each project. Overall, we collected 12,605 version release data points. Each record contains three fields i.e., project name, release version name, release date.

[2] https://developer.atlassian.com/cloud/jira/platform/rest/v2/intro/

[3] https://community.atlassian.com/t5/Jira-questions/..... How-to-retrieve-all-Jira-project-releases-through-API/qaq-p/904556

[4] https://developer.atlassian.com/cloud/jira/platform/rest/v2/api-group-project-versions/#api-rest-api-2-project-projectidorkey-version-get

Table 4.1: The features that we use in our prediction pipeline

Category	**Features from our prior study (Tagra et al.)**	**Our features (JIRA)**	**Type**	**Explanation**
Work habits	Time	Time	Nominal	Time (morning, afternoon, evening, night) during first resolving of the bug.
	Weekday	Weekday	Nominal	Day of the week during first resolving of the bug.
	Month day	Month day	Numeric	Day of the month during first resolving of the bug.
	Month	Month	Numeric	Month during first resolving of the bug.
	Day of year	Day of year	Numeric	Day of year during first resolving of the bug.
Bug report	Component	Component	Nominal	The component of the bug.
	Priority	Priority	Numeric	The priority of the bug.
	Description size	Description size	Numeric	The number of words present in the description.
	Description text	Description text	Text	The text of the description.
	Number of comments	Number of comments	Numeric	The number of comments in the bug report.
	Comments size	Comments size	Numeric	The number of words present in the comments.
	Comments text	Comments text	Text	The text of the comments.
	Priority changed	Priority changed	Boolean	Whether priority changed.
Bug fix	Time days	Time days	Numeric	The time it took to resolve the bug.
	Last status	Last status	Nominal	The last status when the bug was first resolved.
	Number of files in the fix	-	-	-
People	Reporter name	Reporter name	Nominal	The name of the reporter of the bug.
	Assignee name	Assignee name	Nominal	The name of the assignee of the bug.
	Reporter experience	Reporter experience	Numeric	The number of bugs that the reporter has previously reported before reporting this bug.
	Assignee experience	Assignee experience	Numeric	The number of bugs assigned to the assignee before this bug.

4.2.3 Preprocessing JIRA reopened bug dataset features

Figure 4.2 shows the steps that we followed for data collection and preprocessing. Our dataset holds two text features (i.e., *comments text* and *description text*). We preprocessed these text features in line with the approach followed by prior studies [25, 47, 98]. Developer comments and description contain URLs. We substitute them with a representative token to simplify URLs representation [38]. Developers post code blocks in comments and description of bugs. To simplify our text features, we substitute code blocks with a representative token. We process error logs and stack traces similarly by substituting them with a representative token. We also prune uninformative words such as words with length less than three characters and words containing numeric characters. We remove stop words in order to give more focus to important words [87]. We then perform stemming using NLTK package[5] in order to reduce different forms of a word [34, 62, 70, 88]. Finally, we use these two preprocessed text features in addition to the other input features discussed in Table 4.1 to train the model.

4.2.4 Classifying reopened bugs into pre-release reopened bugs and post-release reopened bugs

JIRA bug reopened dataset contains features related to software bugs. JIRA bug also contains a field called *affected version* that determines the software version/s affected by the bug. The information about the release date of the affected version is contained in JIRA version release dataset. We perform an inner join of JIRA bug reopened dataset and JIRA version release dataset using a common key *affected version*. After joining the two datasets, we get the affected version release date for all affected versions for each

[5] https://www.nltk.org/

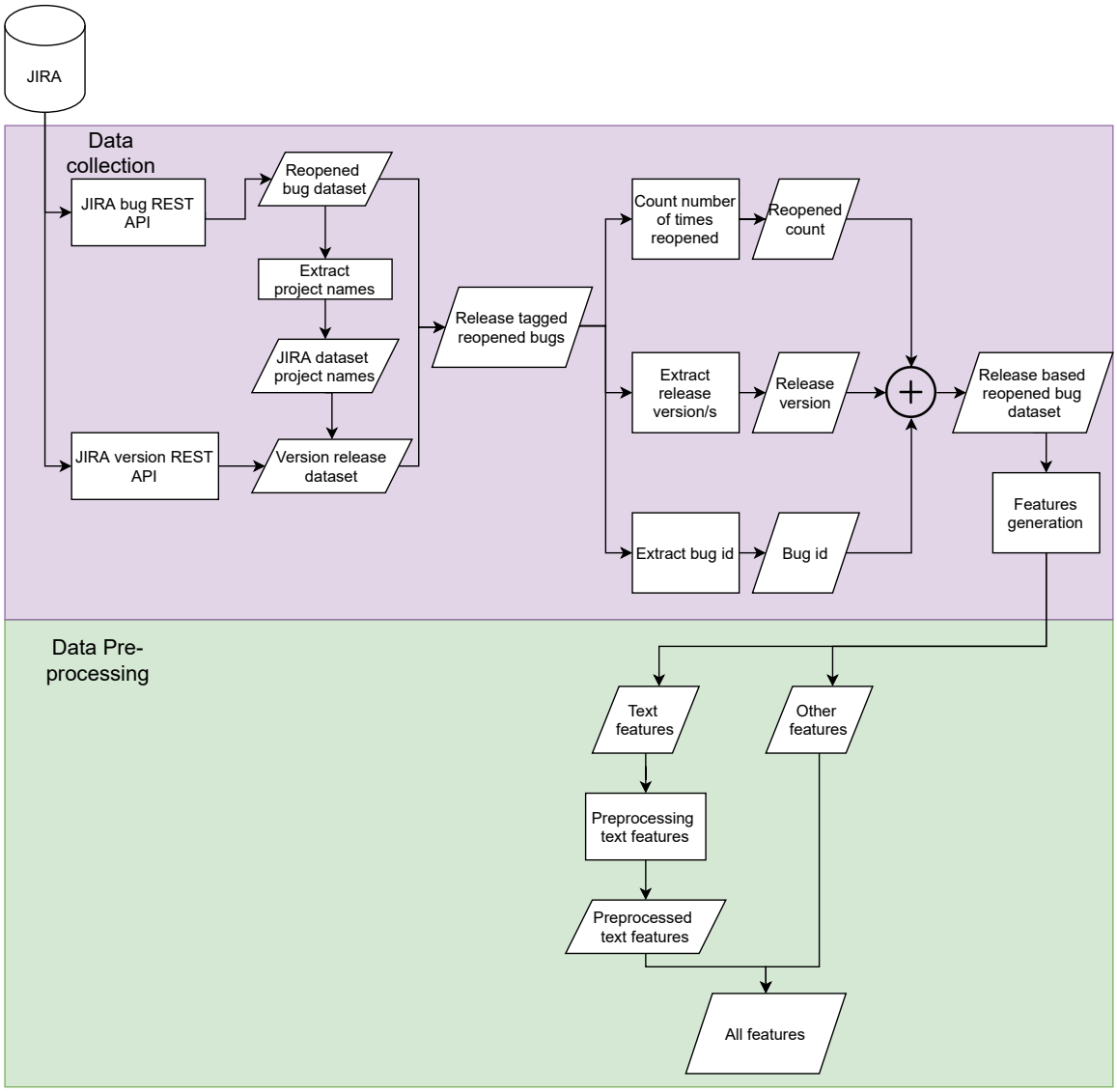

Figure 4.2: Workflow of data collection and preprocessing

bug. A bug can have multiple affected versions and multiple reopens. Hence, we consider a release reopened bug as {bug id, release version, reopened count}. We discard all affected version tag data after the final bug reopen. For an affected version and a reopen, if the affected version is tagged to the bug before it is released, it is classified as a pre-release reopened bug and if the affected version is tagged to the bug after it

is released, it is classified as a post-release reopened bug. For example, in Figure 4.3 a bug has four affected versions v1.1, v1.2, v1.3, and v1.4 and the bug is reopened two times. The bug is a pre-release reopened bug w.r.t v1.1 for first reopened, a pre-release reopened bug w.r.t affected version v1.3 for the first and second reopened, and a post-release reopened bug w.r.t affected version v1.2 for first reopened.

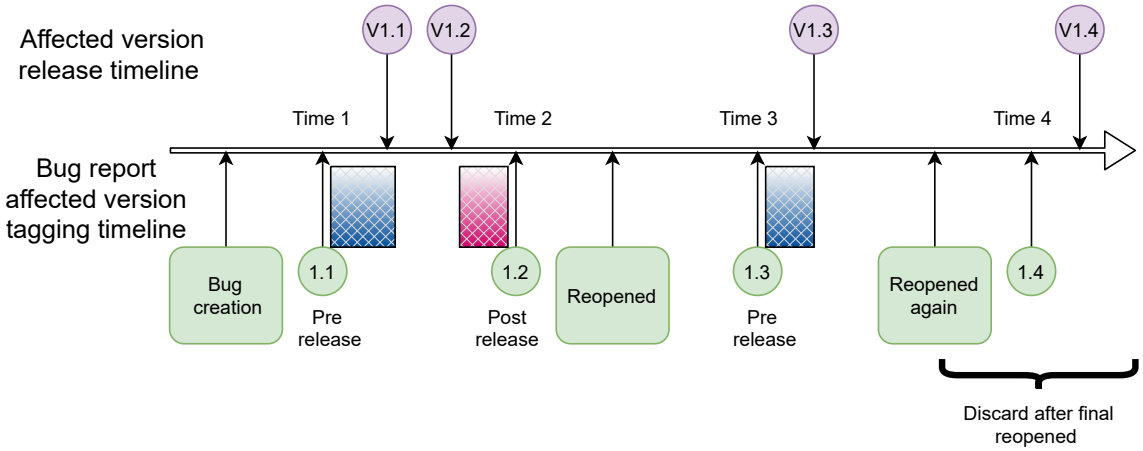

Figure 4.3: Classifying a bug report into pre and post-release reopened bugs.

4.3 Case Study Setup and Results

In this section, we discuss our case study using the reopened bug dataset. We initially study the characteristics of pre-release and post-release reopened bugs in Section 4.3.1. Then we study the prediction of post-release reopened bugs getting resolved rapidly in Section 4.3.2. We bring forth the motivation, approach, and results of all RQs.

4.3.1 RQ1: What are the characteristics of pre-release reopened bugs and post-release reopened bugs?

Motivation: In the software industry pre-release bugs and post-release bugs are handled differently. Taking a cue from the software developers, research also focused on handling pre-release and post-release bugs independently [49, 50, 80]. However, no such distinction is made when it comes to prior studies on reopened bugs. Prior studies considered reopened bugs as a whole [64, 65, 84, 105, 107] without focusing on release based reopened bugs (pre-release reopened bugs and post-release reopened bugs). Therefore, it becomes important to identify whether it is fine to consider reopened bugs as a whole in reopened bugs study or there is a need to categorize reopened bugs into pre-release reopened bugs and post-release reopened bugs. Most prior studies on reopened bugs focused on predicting if a bug will get reopened [83, 84, 105, 106] and identifying the reasons to reopen the bug [120]. However, it is also important to investigate how long a reopened bug takes to finally get resolved. Software developers can leverage such rework time analysis (time spent from when a bug is first reopened to when a bug is finally closed) to identify reopened bugs that took too long to finally resolve. Developers can then go through these bugs to identify reasons for long resolution times of such bugs. Moreover, bugs can get reopened due to various reasons. However, no prior study focused on analysis of the distribution of these reasons to reopen bugs and the rework time of these reasons to reopened bugs on release based reopened bugs (pre-release reopened bugs and post-release reopened bugs). Rework analysis of different reasons to reopen bugs can give insights to development teams to efficiently plan to fix reopened bugs. Therefore, in this RQ, we determine the characteristics of pre-release and post-release reopened bugs based on rework time and

distribution of reasons for reopening the bugs.

Approach: For our analysis we use all 23,350 reopened bugs from the 462 projects that are hosted on JIRA. We perform a quantitative study to identify the characteristics of pre-release reopened bugs and post-release reopened bugs. We then conduct a qualitative study to identify the reasons why pre-release and post-release bugs get reopened. We share the approach followed in the quantitative study and qualitative study below:
Quantitative study: We analyze initial resolving time (initial time) and rework time to resolve release based reopened bugs. We discuss the steps followed in our quantitative study below:

Step 1: Compute the time spent in the initial resolving (i.e., from bug creation to initial bug resolved) and reworking (i.e., from first reopened to final bug resolved) - for all pre, post-release reopened bugs. In this step, we consider only those reopened bugs that are finally resolved. We compute two time spent metrics, these are 1) time spent in the initial resolving - first resolved time minus (–) bug creation time, and 2) time spent in reworking - final bug resolved time minus (–) first bug reopened time. We compute these two metrics for pre-release reopened bugs and post-release reopened bugs. We report the median time for the initial resolving and the reworking for comparison.

Step 2: Compute the difference between rework time and initial time - for all pre, post-release reopened bugs. In this step, we compute the time difference between the reworking and the initial resolving. We report the median time for the initial resolving and reworking for comparison. Since

we are interested in noting if pre-release reopened bugs and post-release reopened bugs are reworked with same urgency as initial effort, we perform a Wilcoxon unpaired test [28] to compare the distributions of difference between rework time and initial resolving time for pre and post-release reopened bugs.

Qualitative study: For our qualitative study we first randomly subsample 100 pre-release reopened bugs and 100 post-release reopened bugs. Our selection of 100 bugs each from both classes of reopened bugs is motivated by a prior study [115] as they also use 100 samples for each category of the studied dataset for their qualitative analysis. We manually study all the associated comments with bug reports. Two researchers (i.e., A1 and A2 with me being one of them) of this study conducted the manual study to tag the reasons why bugs get reopened. After generating the reopened reasons, we compute the time spent in the reworking grouped by reopened reasons. We share the steps followed in our qualitative study below:

Step 1: Generation of tags. A1 randomly selects 25 bugs each from 100 pre-release reopened bugs and 100 post-release reopened bugs. Overall 50 bugs are randomly selected in this step. A1 and A2 independently tag reasons to reopen bugs for these 50 bugs. During this step, the reasons are revised and refined individually.

Step 2: Tagging all bugs. A1 and A2 collect and finalize all the tags representing reasons to reopen bugs. Using these tags they tag all the remaining 150 bugs individually. Both the researchers are encouraged to add new tags if a bug can't be classified by existing tags. It took 8 hours for each researcher to complete this step coding.

Step 3: Computing inter-researcher agreement. We use bug tags to compute the inter-researcher agreement. The inter-researcher agreement is a number between 0 and 1 that is used to determine the agreement reached among two or more researchers in tagging a bug. Increasing the value of inter-researcher agreement increases the overall inter-researcher agreement. We determine the inter-researcher agreement using Cohen's kappa [16, 53, 102]. We computed inter-researcher agreement for tags obtained in Step 2. Our computed inter-researcher agreement for this study is 0.84 (which is substantial) [102].

Step 4: Dealing with disagreements. After tagging all the bugs, we discuss those bugs where A1 and A2 had disagreements for the bug tag. After discussion and mutual agreement, we finalize the tags of disagreed tag bugs.

Step 5: Compute the time spent in reworking - for reopened bugs with reasons. In this step, we consider the reopened reasons identified in Step 2 of the qualitative study. We use the same logic to compute the time spent in reworking as used in Step 1 of quantitative study grouped by reopened reasons. We report the median time for reworking.

Results: 44.2% (i.e., 13,705 out of 31,013) reopened bugs are post-release reopened bugs, whereas only 16.9% (i.e., 5,249 out of 31,013) reopened bugs are pre-release reopened bugs. Using 23,550 reopened bugs from JIRA we observe that there are 31,013 release based reopened bugs {bug id, affected version, reopened count}. Table 4.2 shows the distribution of the types of release based reopened bugs (pre-release reopened bugs and post-release reopened bugs). We observe that 90.4% (i.e., 21,100 out

of 23,350) bugs are reopened once; however, 8.3% (i.e., 1,949 out of 23,350) bugs are reopened twice, and 1.2% (i.e., 291 out of 23,350) bugs are reopened at least three times. Moreover, 37.8% bugs are post-release and are reopened once, however, only 16.2% bugs are pre-release and are reopened once. We also observe that 40.5% (i.e., 9,456 out of 23,350) reopened bugs have unidentified release. For example, in a bug report [6](id: 25013) for the Apache Hive project, the affected version is not tagged in the bug report, hence we classify the bug as belonging to an unidentified release. For every pre-release reopened bug, there are 2.6 post-release reopened bugs in JIRA, indicating that post-release reopened bugs are more common than pre-release reopened bugs.

Table 4.2: Distribution of the types of release based reopened bugs

Reopened bug type	**Reopened bugs {bug id} (%)**	**Release reopened bugs {bug id, affected version, reopened count} (%)**	**Re-opened once (%)**	**Re-opened twice (%)**	**Re-opened 3+ times (%)**
Pre-release	4,164 (17.8%)	5,249 (16.9%)	3,772 (16.2%)	327 (1.4%)	65 (0.3%)
Post-release	9,730 (41.7%)	13,705 (44.2%)	8,816 (37.8%)	795 (3.4%)	119 (0.5%)
Unidentified	9,456 (40.5%)	12,059 (38.9%)	8,522 (36.5%)	827 (3.5%)	107 (0.5%)
Total	23,350 (100%)	31,013 (100%)	21,100 (90.4%)	1,949 (8.3%)	291 (1.2%)

For the pre-release reopened bugs, the median time taken for reworking (i.e., from first reopened to final resolved) is 388.4 hours, whereas for the post-release

[6]https://issues.apache.org/jira/browse/HIVE-25013

reopened bugs, the median time taken for reworking is 189.1 hours. The median time difference between reworking and initial resolving (i.e., from bug creation to initial bug resolved) is 31.9 hours in pre-release reopened bugs as compared to 0.3 hours in post-release reopened bugs. Figure 4.4 shows the distribution of the time spent in various scenarios (i.e., initial resolving, and reworking) for pre-release and post-release reopened bugs. We observe that a pre-release reopened bug takes between 1 minute and 11.1 years of rework time and a post-release reopened bug takes and a post-release bug takes time between 1 minute and 12.2 years. Prior studies [78] find that post-release bugs require twice as large chunks of code changes as compared to pre-release bugs. We observe that while there is only a 10.8 hours median difference of initial resolving in post-release reopened bugs and pre-release reopened bugs, the median time difference of reworking in post-release reopened bugs and pre-release reopened bugs is 149.3 hours. It shows that post-release bugs get more developer attention and a quicker resolution from developers when they are reopened. For example, in a bug report [7] (id: 3369) for the Apache Cloudstack project, the bug is a post-release bug, that gets reopened, and a developer comments *"rajesh can you check on this issue"*, later the assigned developer fixes the bug and closes the bug report. Figure 4.5 shows the distribution of the difference between the time spent in reworking and time spent in initial resolving. The median difference between time spent on reworking and time spent on the initial resolving of a bug is only 0.3 hours in the case of post-release reopened bugs as compared to 31.9 hours in pre-release reopened bugs, indicating that the rework time is almost similar to initial time in case of post-release reopened bugs whereas rework time is higher than initial time in pre-release bugs. The Wilcoxon unpaired test shows that the difference between rework time and initial resolving time

[7] https://issues.apache.org/jira/browse/CLOUDSTACK-3369

for pre-release and post-release is statistically significant with p-value less than 0.05. It shows that post-release reopened bugs are reworked with same urgency as initial resolving.

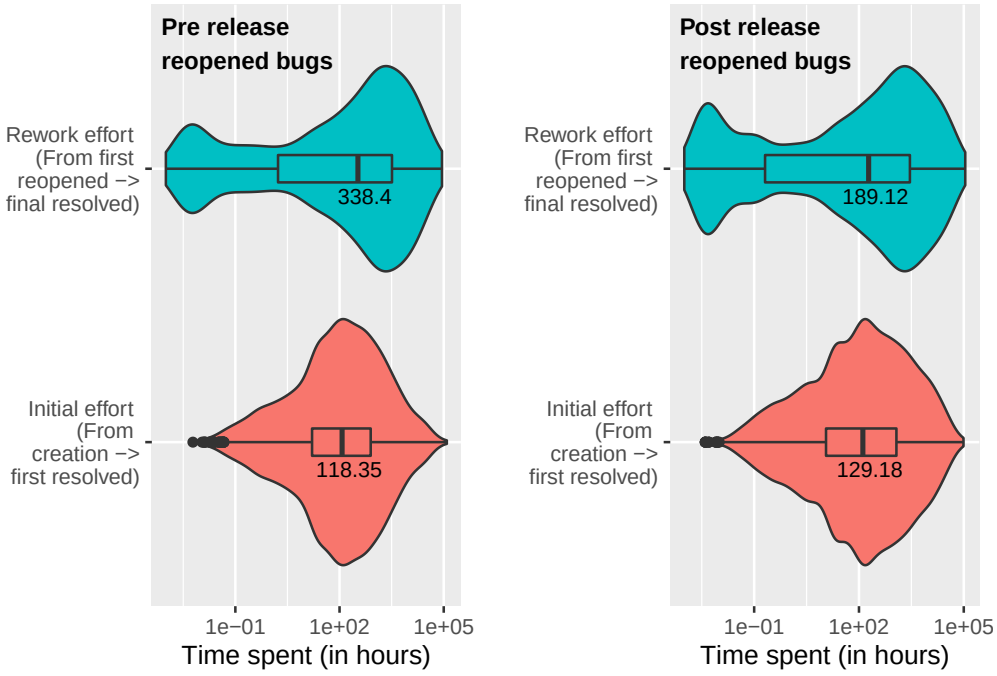

Figure 4.4: Distribution of the time spent in various scenarios for pre-release and post-release reopened bugs.

In our manually studied post-release reopened bugs, technical-patch related issues take only 469.6 hours to rework, however, in pre-release reopened bugs technical-patch related issues take a shorter time (i.e., 198.5 hours) to finally resolve. Furthermore, in post-release reopened bugs technical-integration, and incorrect assessment issues take more than 1,839.3 hours and 4m285 hours respectively to rework as compared to pre-release reopened bugs. Bugs reopened due to documentation issues, take less than eight minutes median time to rework in both pre-release reopened bugs and post-release reopened bugs. From our qualitative analysis, we

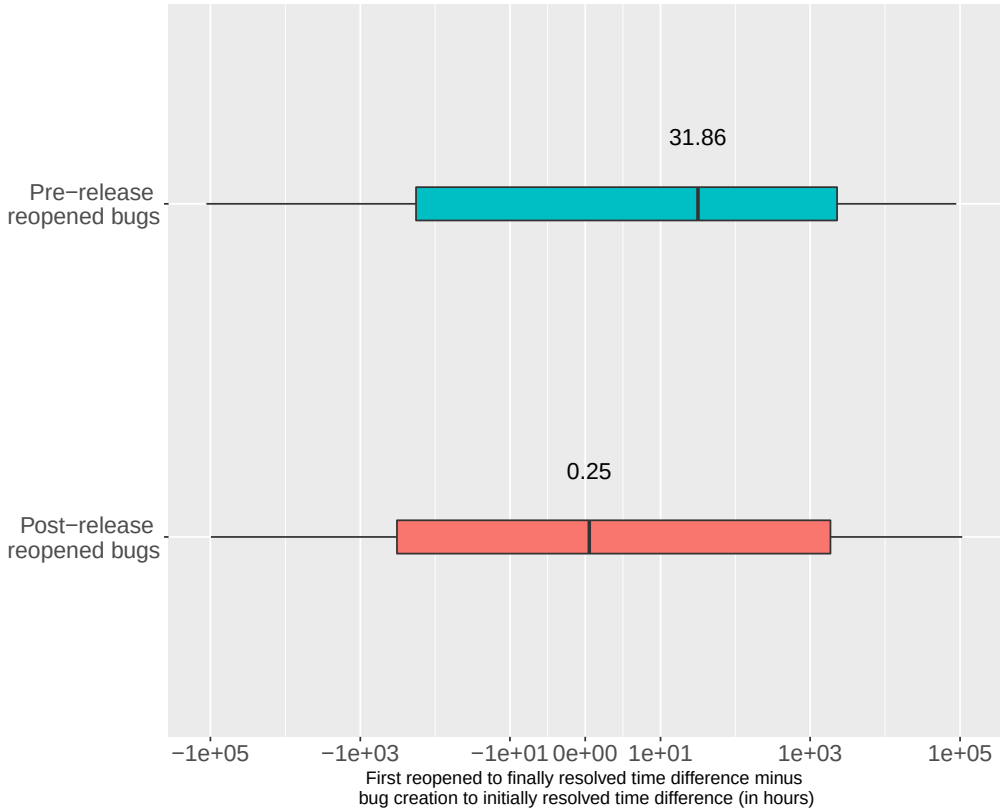

Figure 4.5: Distribution of the difference between the time spent from first bug reopening to final bug resolving and time spent from bug creation to first bug resolving.

observe that bugs get reopened due to four major categories (five reasons), these are technical reasons (i.e., patch issues and integration issues), human reasons (i.e., incorrect assessment), documentation reasons (i.e., documentation issues), and no reason (i.e., no reason). we observe that out of 100 pre-release reopened bugs, 47 are reopened due to patch issues, 21 are due to integration issues, 15 are due to documentation issues, 8 are due to incorrect assessment, whereas 9 are due to unidentified reasons. Out of 100 post-release reopened bugs 46 are reopened due to patch related issues, 22 are due to documentation issues, 21 are due to incorrect assessment, 10 are due to incorrect assessment and 1 is due to unidentified reasons. Figure 4.6 shows the distribution

of rework time for various reasons of reopening a bug in pre-release and post-release reopened bugs. Bugs reopened due to incorrect assessment take the most rework time for post-release reopened bugs. For example, in a bug report [8] (id: 7646) for the Apache CXF project, the bug is post-release for affected version 3.0.3 and is reopened due to incorrect assessment, since this bug was a duplicate of another bug and both bugs were closed. During reopening the bug, a developer comments *"Hi, Both CXF-7645 and CXF-7646 are same. But mentioned both requests as duplicate and closed both the request. Kindly consider this request (re-opening) and help me."* It took 7 months to finally resolve the bug from when it was reopened for the first time.

Moreover, we observe that in post-release reopened bugs, patch issues take only 469.6 hours to rework, whereas technical-integration and incorrect assessment issues take 2,173.1 hours and 4,848.2 hours respectively to rework. It indicates that in post-release reopened bugs, technical-patch related issues are faster to rework whereas technical-integration issues and incorrect assessment issues are more challenging to rework. Moreover, post-release bugs reopened due to technical-patch related issue take 271.1 hours more to rework as compared to pre-release reopened bugs. Furthermore, bugs reopened due to documentation issues are resolved within a median of eight minutes median time in both pre-release reopened bugs and post-release reopened bugs, indicating that documentation issues do not require much time to rework for pre as well as post-release reopened bugs. It indicates that, pre-release reopened bugs and post-release reopened bugs have different resolving speeds based on reason to reopen bugs. More specifically, technical-patch related bugs are finally resolved faster in post-release bugs, whereas, technical-integration and incorrect assessment reopened bugs are finally resolved faster in pre-release reopened bugs. Future studies on estimating

[8] https://issues.apache.org/jira/browse/CXF-7646

time to fix reopened bugs should ignore bugs reopened due to documentation issues, since they are fixed really fast and without any technical/code related change no matter whether they are pre-release or post-release reopened bugs. These bugs should be considered a noise for reopened bug fixing activity.

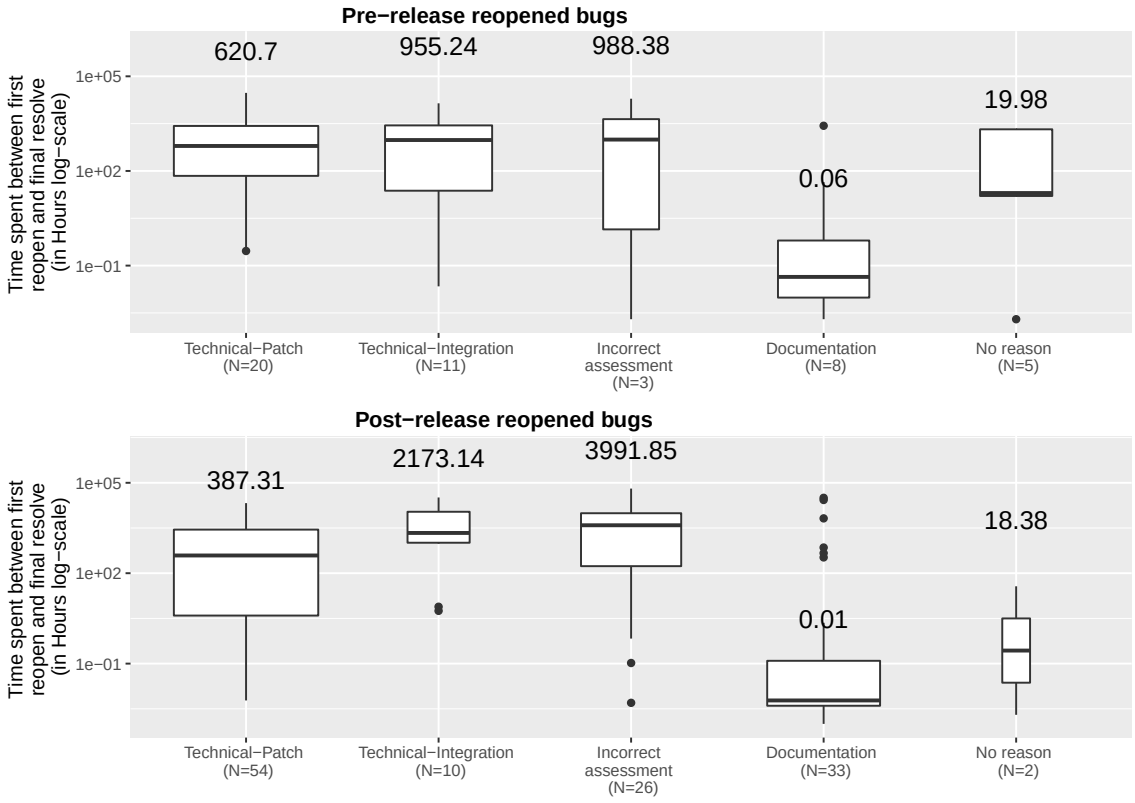

Figure 4.6: Distribution of bug rework time for various reasons of reopening a bug in pre-release and post-release reopened bugs.

Summary of RQ1

For every pre-release reopened bug, there are 2.6 post-release reopened bugs in JIRA, indicating that there are more post-release bugs than pre-release bugs. The median difference between time spent in reworking and time spent in the initial resolving of a bug is only 0.3 hours in case of post-release reopened bugs as compared to 31.9 hours in pre-release reopened bugs. Technical-patch, technical-integration and incorrect assessment reopened bugs are finally resolved slower in post-release bugs as compared to pre-release reopened bugs. Out of all reasons to reopen bugs, only documentation issues do not require much time (a median of less than four minutes) to rework pre as well as post-release reopened bugs.

4.3.2 RQ2: Can models predict if a post-release reopened bug will get resolved rapidly/slowly with an acceptable performance?

Motivation: Rwemalika et al. [78] discuss that post-release bugs require at least twice bigger chunks of changes as compared to pre-release bugs. They also discuss that research on software systems should consider post-release bugs for their studies as post-release bugs are harder to reveal as compared to pre-release bugs. Prior studies [23, 50, 52, 113] focused on post-release bugs for their analysis. Post-release bugs gather more developer attention than pre-release bugs that's why research has focused on studying various aspects of post-release bugs such as their fixing time [49, 50, 80]. We observe in RQ1, that pre-release reopened bugs and post-release reopened bugs are completely different in a plethora of ways. However, reopened bugs studies take pre-release and post-release reopened bugs together [64, 65, 83, 84, 106, 107]. However,

this is in contrast with traditional defect prediction studies where researchers don't consider pre-release bugs and post-release bugs together [49, 50, 80]. Prior studies in software bugs are focused on predicting the issue lifetime of a bug (including how fast a bug can be resolved) [31, 51, 59, 72, 76]. However, as per our knowledge, no prior study focused on predicting the resolving time of reopened bugs. It is important to determine if a reopened bug will be resolved rapidly or slowly. If we can predict well whether a post-release reopened bug will get fixed rapidly or slowly, we can notify developers about those post-release reopened bugs that will take a high time to finally resolve. Post-release reopened bugs are more critical to resolve fast as they affect a very large audience (i.e., software clients). Developers can then allocate more effort and resources towards resolving post-release reopened bugs with slow resolving time and allocate more resources towards those reopened bugs. Moreover, we also interpret the prediction pipeline to understand what characteristics one can look for to identify if a post-release reopened bug will get fixed fast or slow.

Approach: We construct prediction pipelines to predict if a post-release reopened bug will get resolved rapidly/slowly. Figure 4.7 the prediction pipeline that we use to predict if a post-release reopened bug will get resolved rapidly/slowly. We discuss the steps followed to construct the prediction pipeline below:

> **Step 0: Select data points and label the data points of the generated dataset for prediction.** From the whole post-release reopened dataset, we only consider unique bugs that were finally resolved. We sort the dataset based on the rework time (i.e., time spent from when a bug is reopened for the first time to when the bug is finally resolved). We label the top 20% bugs as fast

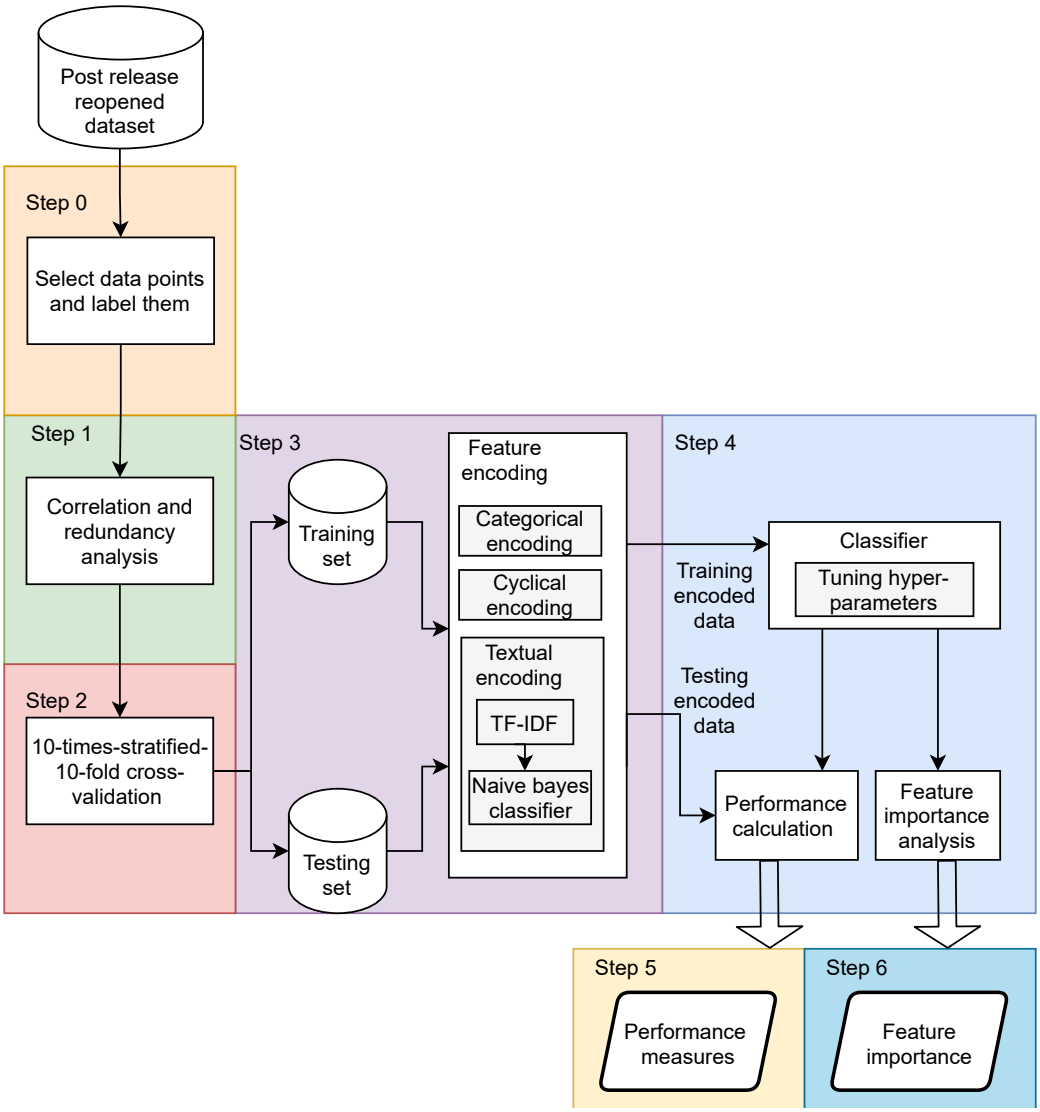

Figure 4.7: A model to predict if a post-release reopened bug will get resolved rapidly/slowly.

resolved and the bottom 20% bugs as slow resolved based on the rework time. We follow the same approach followed by prior work to select top and bottom bugs [54, 74, 93]. Since all our studied features in post release reopened bugs dataset have the same values for various affected versions tagged in a bug and various reopened counts, we only consider bugs with

unique bug id in our prediction pipeline in order to avoid duplicate data points in our training set. For our analysis, we only consider those bugs that are finally resolved. Note that, in this step since we select equal samples of each label for our study, we don't encounter class imbalance issue in our prediction pipeline. We call this Step 0 because this step is used for data generation, it is a prerequisite for the prediction pipeline, and Step 1 – Step 6 are used for constructing the prediction pipeline.

Step 1: Correlation and redundancy analysis. Tantithamthavorn et al. [90] argue that correlated features interfere with each other at the time model interpretation. Moreover, Jiarpakdee et al. [42, 44] asserts that correlated metrics impact feature importance ranks of the prediction pipeline. Therefore taking a cue from the prior studies [24, 42, 43, 54, 60, 73, 74, 90], we carry out a correlation and redundancy analysis on all the independent features belonging to our dataset. We remove correlated and redundant features using Autospearman method provided by Rnalytica[9] R package [54, 90, 113]. For the Spearman correlation coefficient, we use a default value of ρ as a threshold.

Step 2: Splitting dataset into training/testing sets. We use 10-times-stratified 10-fold cross-validation for splitting the dataset into training and testing datasets. In this method, we split the whole dataset into 10 equal folds where each fold contains an equal percentage of all class labels (in our case fast resolved and slow resolved). After this, we use each fold to iteratively test the performance of the model and the remaining 9 folds to train the

[9] https://rdrr.io/github/software-analytics/Rnalytica/man/AutoSpearman.html

model. Then we repeat this process 10 times. After this step, 100 performance and feature importance measures are generated (i.e., 10 for each of 10 folds). More details of stratified k-fold cross-validation can be found by referring to Zeng et al. [116].

Step 3: Feature encoding.

Step 3.1 Categorical features encoding. Our dataset contains categorical features such as *reporter name* and *last status*. We use one-hot feature encoding to encode categorical features into numerical features in our dataset in order for them to be used as input for training the model [119]. We use one-hot encoding for transforming categorical features into numeric features as prior studies [4, 17, 40] use one-hot encoding for transforming categorical features into numerical features. For each categorical feature, one-hot encoding generates k boolean columns for k category values of that feature, with only one column representing that category value is set (i.e., 1) and the rest all columns are unset (i.e., 0).

Step 3.2 Cyclical features encoding. For the feature *weekday*, we perform cyclical encoding as *weekday* feature is cyclic. The range of transformed cyclical features is from -1 to 1. Note that we use cyclical features encoding for *weekday* features as input to train the model with all learners except Multinomial Naive Bayes learner because Multinomial Naive Bayes learner requires that all features ranges should be greater than zero [77]. Therefore, only for Multinomial Naive Bayes learner, we use one-hot encoding even for *weekday* feature.

Step 3.3 Textual features encoding. For the text features such as *comments*

text and *description text*, we use TF-IDF (i.e., term frequency-inverse document frequency) vectorization [71] similar to prior studies [12, 61, 71, 75]. TF-IDF assigns a higher weight to rare words (words that appear in very few documents). Also, If a word appears multiple times in a document, TF-IDF assigns it a higher weight. If a word appears in almost all documents, TF-IDF assigns it the lowest weight [21]. For each text feature, we use training data to train TF-IDF model and use the trained TF-IDF model to generate the text scores.

After generating the text feature scores, we use Naive Bayes learner to compute text probabilities similar to the prior study [63, 84, 117]. Using only TF-IDF for handling text features will generate a very high number of features (one feature for each unique word in the text document) that are even more than the number of samples in the dataset. This is known as the curse of dimensionality [100] and it leads to overfitting in the model. Moreover, when we interpret the models generated using only TF-IDF generated features, it will demonstrate the importance of words in the prediction pipeline. However, we are interested in determining if the text features such as *comments text* and *description text* are among the important features (rather than determining important words) in the model prediction. Hence, we train two Naive Bayes learners (one each for *comments text* and *description text*) on the training data. These two generated text probabilities are used as two of the input features for predicting if a reopened bug will get resolved rapidly/slowly.

Step 4: Model construction.

Step 4.1 Training the model. We use all the training features to train the model for prediction. We use 7 most commonly used learners for our study. We show the hyperparameters and learners used for our study in Table 4.3. We choose these 7 learners for our study because a prior study by Agrawal et al. [1] finds that KNN, Decision Tree, Logistic Regression, Random Forest, and Multinomial Naive Bayes are widely used learners in software analytics. Moreover, we also include Adaboost, and Gradient Boosting in our study since these learners are also widely used in prior studies related to software analytics [29, 90, 91].

Step 4.2 Hyperparameters tuning. For our model, we use grid search based hyperparameters tuning [81]. Grid search uses the brute force approach by selecting all the combinations of parameter values used to train the model and selecting those values which give the best results (we consider AUC). We perform hyperparameters tuning on our model as several prior studies discuss that hyperparameters tuning plays a vital role to ensure optimal performance and interpretation of the model [27, 90, 91].

Step 5: Model performance evaluation. We calculate the performance of all learners on predicting if a reopened bug will get resolved rapidly/slowly. For our study, we choose AUC to calculate the performance of learners because AUC is threshold-independent. Prior studies advocate using AUC performance measure to assess the performance of prediction pipelines [30, 55, 73, 74, 91]. We consider the performance of a model as acceptable if the model gives median AUC across 100 performance iterations greater than 0.7 [3, 45, 66, 97]. Otherwise, we consider the model performance as

Table 4.3: Overview of the used hyperparameters in our study. The default values are shown in bold. Randuniform denotes drawing random samples from a uniform distribution, and intranduniform denotes denotes drawing random samples from a uniform distribution and converting them to their nearest integer value.

Learner	**Parameter name**	**Parameter description**	**Parameter range**
Random Forest	n_estimators	The number of trees in the forest.	(**100**, randuniform(50, 150))
	criterion	The function for measuring the split quality.	(**gini**, entropy)
	min_samples_split	The minimum samples needed to split a node.	(**2**,randuniform(0.0, 1.0))
Gradient Boosting	n_estimators	The number of boosting stages to perform.	(**100**, randuniform(50, 150))
	min_samples_leaf	The minimum samples needed to be at a leaf node.	(**1**, randuniform(1, 10))
Logistic Regression	penalty	To specify the norm used in the penalization.	(l1,**l2**)
	tol	The tolerance for the stopping criteria.	(**1e-4**, randuniform(0.0,0.1))
	C	The inverse of regularization strength.	(**1**, randint(1,500))
Adaboost	n_estimators	The maximum number of estimators at which the boosting is terminated.	(**50**, randuniform(50, 150))
Multinomial Naive Bayes	alpha	The additive smoothing parameter.	(**1**, randuniform(0, 1))
Decision Tree	splitter	Choosing strategy for splitting at each node.	(**best**, random)
	criterion	The function for measuring the split quality.	(**gini**, entropy)
	min_samples_split	The minimum samples needed to split a node.	(**2**, randuniform(0.0,1.0))
KNN	n_neighbors	The number of neighbors.	(**5**, randint(2, 25))
	weights	The weight function used in prediction.	(**uniform**, distance)
	p	Power parameter.	(**2**,randint(1,15))
	metric	The distance metric to use for the tree.	(**minkowski**, chebyshev)

poor.

Step 6: Ranking important features. we employ the permutation feature importance method [5] to generate the feature importance in predicting if a reopened bug will get resolved rapidly/slowly. Permutation feature importance method works as follows: We calculate the performance of the prediction pipeline using all features. We then randomly permute the values of

each feature at a time and calculate the performance drop of the prediction pipeline. The more the performance drop due to a feature value change, the higher is the importance of that feature for the prediction pipeline.

We calculate the feature importance ranks only for those learners that give an acceptable AUC performance ($\geqslant 0.7$) as Lipton et al.; Chen et al. [19, 57] assert that to interpret a model, it should have an acceptable performance. After this, importance scores for each feature and for each of 100 iterations are generated. Details about the working of permutation feature importance can be obtained at Altmann et al. [5]. We then generate feature importance ranks from feature importance scores using Scott-Knott ESD test [92]. These generated feature importance ranks determine the order of feature importance for the prediction pipeline. We then note down the top three important features for the prediction pipeline.

Results: At best we are able to predict if a post-release reopened bug will get resolved rapidly/slowly with an AUC of 0.78. In RQ1, we observe that there are 13,705 unique post-release reopened bugs (i.e., {bug id, affected version, reopened count}). For prediction pipeline, we observe that there are 9,730 post-release reopened bugs with unique bug ids, out of these 27 bugs have missing data, so we ignore these bugs. We only consider those bugs that are finally resolved. We observe that 93.1% (i.e., 9,031 out of 9,703) post-release reopened bugs are finally resolved. We consider the top 20% of these bugs as fast resolved and the bottom 20% of these bugs slow resolved based on the rework time. Hence, we obtain 1,806 post-release fast resolved reopened bugs and

1806 post-release slow resolved reopened bugs. We observe that fast resolved post-release reopened bugs take in between 1 minute and 3.3 minutes to get resolved and slow resolved post-release reopened bugs take in between 4,538 hours and 107,002 hours. Therefore, we consider these 3,612 post-release reopened bugs in our prediction task. We observe that we are able to predict if a post-release reopened bug will get resolved rapidly/slowly with a maximum AUC of 0.78. Prior studies [3, 45, 66, 97] show that AUC $\geqslant$0.7 is considered as an acceptable performance. Figure 4.8 shows the distribution of AUC for various learners (i.e., classifiers) to predict if a post-release reopened bug will get resolved rapidly/slowly.

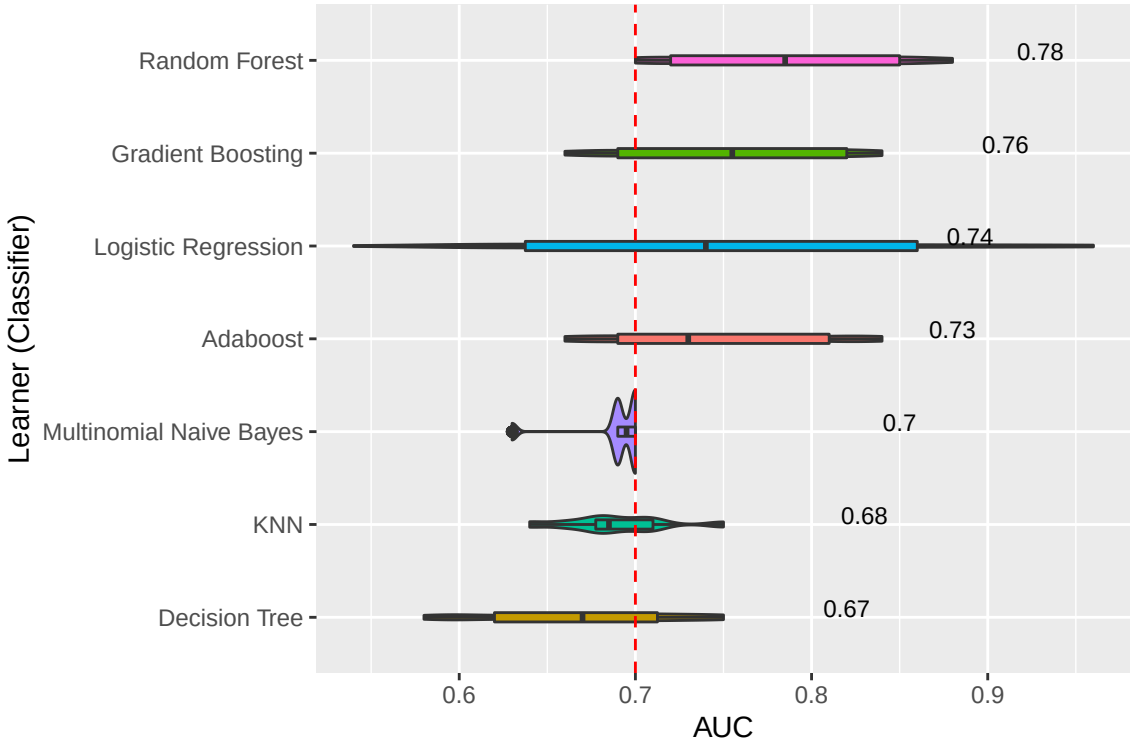

Figure 4.8: Distribution of AUC for various learners (i.e., classifiers) and median AUC scores to predict if a post-release reopened bug will get resolved rapidly/slowly.

Out of 7 studied learners (i.e., classifiers), 5 learners yield an acceptable AUC (⩾0.7) to predict if a post-release reopened bug will get resolved rapidly/slowly. These are Random Forest, Gradient Boosting, Logistic Regression, Adaboost, and Multinomial Naive Bayes. Figure 4.8 shows the distribution of AUC for various learners (i.e., classifiers) to predict if a post-release reopened bug will get resolved rapidly/slowly. Random forest learner gives the highest AUC i.e., 0.78 among other learners. We observe that 5 out of 7 learners give acceptable AUC to predict if a post-release reopened bug will get resolved rapidly/slowly. Only KNN and Decision Tree learners yield an AUC of 0.68, and 0.67 respectively which is considered as poor AUC (i.e., AUC<0.7).

***Comments size, comments text*, and *description text* are among the top three important features in 100%, 80%, and 60% of learners with an acceptable AUC (⩾0.7) respectively.** Table 4.4 shows the top three feature importance ranks for various learners (i.e., classifiers) to predict if a post-release reopened bug will get resolved rapidly/slowly. We only consider Random Forest, Gradient Boosting, Logistic Regression, Adaboost, and Multinomial Naive Bayes learners for feature importance analysis, since only these 5 learners give an acceptable AUC to predict if a post-release reopened bug will get resolved rapidly/slowly. We observe that *comments size*, and *comments text* are among the top three most important features in 100% and 80% of learners respectively to predict if a post-release reopened bug will get resolved rapidly/slowly. For example, in a post-release reopened bug report[10] (id: 8725) for the Apache SOLR project, a developer comments *"Can you close the issue as it is commited on the 5.5 branch?"*, later the bug is finally resolved within two hours. Since *comments text* is among the most

[10] https://issues.apache.org/jira/browse/SOLR-8725

important features to predict if a post-release reopened bug will get resolved rapidly/slowly, future research can leverage developer comments and analyze the reasons why some post-release reopened bugs take too long to get finally resolved.

Table 4.4: Top three feature importance ranks for various learners (i.e., classifiers) to predict if a post-release reopened bug will get resolved rapidly/slowly

Learner (Classifier)	**Type of model**	**Feature**	**Rank**	**AUC**
Random Forest	Post-release reopened	Description text	1	0.78
		Comments text	2	
		Comments size	3	
Gradient Boosting	Post-release reopened	Comments text	1	0.76
		Description text	2	
		Comments size	3	
Logistic Regression	Post-release reopened	Reporter experience	1	0.74
		Comments text	2	
		Comments size	3	
Adaboost	Post-release reopened	Comments text	1	0.73
		Description text	2	
		Comments size	3	
Multinomial Naive Bayes	Post-release reopened	Comments size	1	0.7
		Time days	2	
		Description size	3	

Summary of RQ2

Predicting if a post-release reopened bug will get resolved rapidly/slowly is trustworthy, since it gives an acceptable AUC (⩾0.7) of 0.78. 5 out of 7 learners (i.e., Random Forest, Gradient Boosting, Logistic Regression, Adaboost, and Multinomial Naive Bayes) give acceptable AUC to predict if a post-release reopened bug will get resolved rapidly/slowly. Features generated from the developer comments such as *comments text*, and *comments size* and developer added bug description such as *description text* are important features to predict if a post-release reopened bug will get resolved rapidly/slowly.

4.4 Discussion

4.4.1 Do models trained on pre-release reopened bugs have similar performance as models trained on post-release reopened bugs while predicting if a reopened bug will get resolved rapidly/slowly?

Motivation: From the results of RQ1 (Section 4.3.1) we observe that pre-release reopened bugs and post-release reopened bugs have different characteristics in terms of rework time and distribution of reopened reasons. Rwemalika et al. [78] discusses that research on software systems should consider post-release bugs for their studies as post-release bugs are harder to reveal as compared to pre-release bugs. Therefore, in RQ2 (Section 4.3.2) we construct a prediction pipeline to predict if a post-release reopened bug will get fixed rapidly? We observe that we are able to achieve an AUC of

0.78 (which is acceptable) for a prediction pipeline. We also observe that *comments text*, *comments size*, and *description size* are the topmost features to predict if a post-release reopened bug will get resolved rapidly/slowly. We agree that post-release reopened bugs are more critical to predict if they will get resolved rapidly/slowly than pre-release bugs. However, it becomes important for us to understand how the performance of a model for predicting if a pre-release reopened bug will get fixed rapidly will compare against performance for post-release reopened bugs model. This comparison is important because most studies involving reopened bugs do not consider post-release reopened bugs and pre-release reopened bugs separately. The magnitude of difference between performance results obtained using pre-release reopened bugs and post-release reopened bugs will drive future studies to revisit reopened bugs studies by segregating pre-release reopened bugs and post-release reopened bugs.

Approach: We follow the same approach as followed in RQ2 (Section 4.3.2) to construct a model pipeline for predicting if a release based reopened bug will get resolved rapidly/slowly. The only difference is that here we consider pre-release reopened bug dataset instead of post-release reopened bug dataset used in RQ2.

Results: At best we are able to predict if a pre-release reopened bug will get resolved rapidly/slowly with an AUC of 0.72 which is 6% less than using post-release reopened bugs for our model. In RQ1, we observe that there are 5,249 unique pre-release reopened bugs (i.e., {bug id, affected version, reopened count}). For our prediction pipeline, we observe that there are 4,164 pre-release reopened bugs with unique bug ids. Out of these 4,164 pre-release reopened bugs, 27 bugs have missing data, so we ignore these

bugs. For our analysis, we only consider those bugs which are finally resolved. We observe that 93.2% (i.e., 3,855 out of 4,137) pre-release reopened bugs are finally resolved. We consider the top 20% of these bugs as fast resolved and the bottom 20% of these bugs slow resolved based on the rework time. Hence, we obtain 771 fast resolved pre-release reopened bugs and 771 slow resolved pre-release reopened bugs. We observe that fast resolved pre-release reopened bugs take in between 1 minute and 13.2 minutes to get resolved and slow resolved pre-release reopened bugs take in between 5,528 hours and 97,126 hours. Hence we consider these 1,542 pre-release reopened bugs in our prediction task. Prior studies [3, 45, 66, 97] show that AUC $\geqslant$0.7 is considered as an acceptable performance. We observe that we are able to predict if a pre-release reopened bug will get resolved rapidly/slowly with a maximum median AUC of 0.72. Figure 4.9 shows the distribution of AUC for various learners (i.e., classifiers) and median AUC scores to predict if a pre-release reopened bug will get resolved rapidly/slowly.

Out of 7 studied learners (i.e., classifiers), only Random Forest and Logistic Regression yield an acceptable AUC ($\geqslant$0.7) to predict if a pre-release reopened bug will get resolved rapidly/slowly. Figure 4.9 shows the distribution of AUC for various learners (i.e., classifiers) to predict if a pre-release reopened bug will get resolved rapidly/slowly. Random Forest and Logistic Regression yield an AUC of 0.72, and 0.7 respectively which is acceptable ($\geqslant$0.7). However, Adaboost, Gradient Boosting, Multinomial Naive Bayes, Decision Tree, and KNN yield an AUC of 0.69, 0.68, 0.66, 0.65, and 0.61 respectively which is considered as poor AUC (i.e., AUC<0.7). This is in contrast with the findings of the prediction pipeline involving post-release reopened bugs, where it is observed that 5 out of 7 learners yield an acceptable AUC to predict if a post-release reopened bug will get resolved rapidly/slowly.

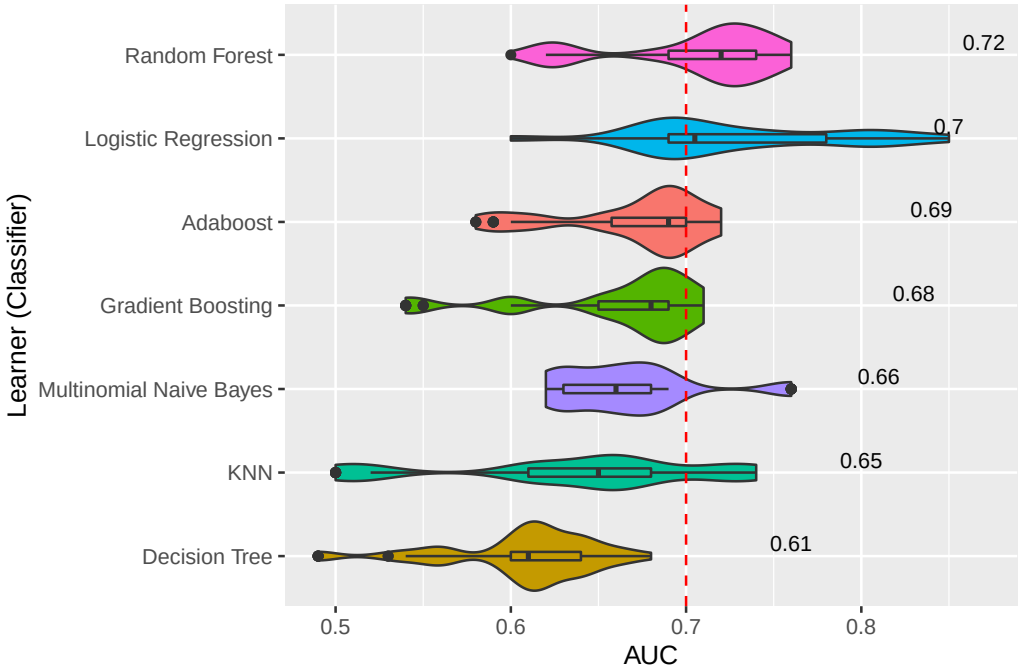

Figure 4.9: Distribution of AUC for various learners (i.e., classifiers) and median AUC scores to predict if a pre-release reopened bug will get resolved rapidly/slowly

***Comments text* (i.e., developer comments) and *number of comments* are among the top three features to predict if a pre-release reopened bug will get resolved rapidly/slowly in all the learners with an acceptable AUC (⩾0.7).** Table 4.5 shows the top three feature importance ranks for various learners (i.e., classifiers) to predict if a pre-release reopened bug will get resolved rapidly/slowly. We only consider Random forest and Logistic Regression learners for feature importance analysis, since only these two learners give an acceptable AUC to predict if a pre-release reopened bug will get resolved rapidly/slowly. We observe that *comments text*, and *number of comments* are the most important features to predict if a pre-release reopened bug will get resolved rapidly/slowly. For example, in a pre-release reopened bug report[11] (id: 286) for the

[11] https://issues.apache.org/jira/browse/CLOUDSTACK-286

Apache Cloudstack project, a developer comments *"Any progress on this?"*, later the bug is finally resolved within 3 days. However, this finding is in contrast with the findings in case of post-release reopened bugs (in RQ2) where *number of comments* was not observed as an important feature to predict if a post-release reopened bug will get resolved rapidly/slowly.

Since *comments text* is among the most important features to predict if a pre-release reopened bug will get resolved rapidly/slowly, future research can leverage developer comments and analyze the reasons why some pre-release reopened bugs take too long to get finally resolved.

Table 4.5: Top three feature importance ranks for various learners (i.e., classifiers) to predict if a pre-release reopened bug will get resolved rapidly/slowly

Learner (Classifier)	**Type of model**	**Feature**	**Rank**	**AUC**
Random Forest	Pre-release reopened	Comments text	1	0.72
		Number of comments	2	
		Description text	3	
Logistic regression	Pre-release reopened	Number of comments	1	0.7
		Comments text	2	
		Time days	3	

Table 4.6: Our major findings and their implications.

Audience	**Findings**	**Implications**
Developers	20.8% of post-release reopened bugs are reopened due to incorrect assessment.	Developers should prioritize their effort on a correct assessment of the bug resolution, especially regarding bug reproducibility.
Researchers	26.4% of the post-release reopened bugs are reopened due to documentation issues. Bugs reopened due to documentation issues do not add any technical knowledge about software bugs. They are just for the purpose of updating the documentation of the software.	Researchers should develop a classifier to classify reasons for reopening a post-release bug and use that classifier to filter out post-release bugs reopened bugs due to documentation issues from their study of computing post-release reopened bugs rework effort.
	Post-release reopened bugs give an AUC of 0.78 for predicting if a release based reopened bug will get resolved rapidly/slowly, which is 6% greater than AUC obtained for the same model using pre-release reopened bugs. 5 out of 7 learners give an acceptable AUC for the prediction pipeline using post-release reopened bugs whereas only 2 out of 7 learners give an acceptable AUC for the prediction pipeline.	Prior studies on reopened bugs should revisit reopened bugs while considering post-release reopened bugs for their study.
	Both the Random Forest, and Logistic Regression learners give an acceptable AUC to predict if a release based reopened bug will get resolved rapidly/slowly for both pre-release and post-release bugs.	Researchers should use Random Forest, and Logistic Regression in their study related to the prediction of release based reopened bugs getting fixed rapidly to get the best performance.
	Features related to developer comments are important in predicting if a release based reopened bug will get resolved rapidly/slowly.	Developer comments can lead researchers to understanding the reasons why release based bugs can sometimes take too much time to resolve.

4.4.2 The implications of our study

Implications for the developers:

Developers should prioritize their effort on the correct assessment of the bug resolution, especially regarding bug reproducibility, since we observe that 20.8% of post-release reopened bugs are reopened due to incorrect assessment. Since post-release bugs affect a very large audience (i.e., clients), it becomes obvious to prioritize handling post-release reopened bugs on priority. Moreover, out of all the post-release reopened bugs, those due to incorrect assessment take the maximum rework time (i.e., 4,848.2 hours). For example, in a post-release bug[12] (id: 5071) for the project Apache Wicket, a developer initially was not able to reproduce the issue, but later he realized that he can reproduce the issue. His comment *"I'm able to reproduce it now. Let me see..."* demonstrates that he had incorrectly assessed the reopened issue as non reproducible, but later it was observed that the bug is reproducible.

Implications for the researchers:

Researchers should develop models to classify the reasons for reopening a post-release bug and use such models to filter out post-release reopened bugs due to documentation issues from their study of computing post-release reopened bugs rework effort. Bugs reopened due to documentation issues do not add any technical knowledge about software bugs. They are just for the purpose of updating the documentation of the software. We observe in RQ1 from a sample of 100 post-release reopened bugs that post-release bugs reopened due to documentation issues make up 22% of

[12] https://issues.apache.org/jira/browse/WICKET-5071

all post-release reopened issues and have a rework time of just under a minute. Incorporating the post-release bugs reopened due to documentation with post-release reopened bugs due to other technical issues can underestimate the median rework effort for post-release reopened bugs. Hence, we recommend researchers develop a post-release reopened reason classifier and filter out post-release bugs reopened due to documentation issues.

Prior studies on reopened bugs should revisit reopened bugs while considering post-release reopened bugs for their study. Traditional studies on bugs focus on pre-release bugs and post-release bugs seperately. Moreover, Rwemalika et al. [78] discusses that post-release bugs require at least two times bigger chunks of changes as compared to pre-release bugs, and 66.7% of the post-release bugs require changes in configuration files as compared to only 46% configuration file changes in case of pre-release bugs. However, no prior studies on reopened bugs consider post-release reopened bugs specifically for their study. We observe in Section 4.3.1 that pre-release reopened bugs and post-release reopened bugs have different characteristics based on time spent to resolve a reopened bug. Furthermore, we observe in Section 4.3.2 that post-release reopened bugs give an AUC of 0.78 for predicting if a release based reopened bug will get resolved rapidly/slowly, which is 6% greater than AUC obtained for the same model using pre-release reopened bugs. In addition, we observe that 5 out of 7 learners give an acceptable AUC for the prediction pipeline using post-release reopened bugs whereas only 2 out of 7 learners give an acceptable AUC for the prediction pipeline. It is obvious from the above discussion that post-release reopened bugs

are more critical to analyze and also post-release reopened bugs give better performance in prediction tasks and post-release reopened bugs work well more varied set of learners. Studies on reopened bugs that do not segregate post-release reopened bugs from pre-release reopened bugs suffer from bias. We suggest researchers revisit prior studies on reopened bugs to consider post-release reopened bugs for studies such as prediction of reopened bugs [83, 84, 105, 106].

Researchers should use Random Forest and Logistic Regression learners in their study related to the prediction of release based reopened bugs getting fixed rapidly to get the best performance. We observe in Section 4.3.2 and Section 4.4 that both the Random Forest, and Logistic Regression learners give an acceptable AUC to predict if a release based reopened bug will get resolved rapidly/slowly for both post-release reopened bugs and pre-release reopened bugs.

Developer comments can lead researchers to understanding the reasons why release based bugs can sometimes take too much time to resolve. In Section 4.4 we observe that *comments text* and *number of comments* are among the top three features to predict if a pre-release reopened bug will get resolved rapidly/slowly. Moreover, in Section 4.3.2 we observe that *comments size* and *comments text* are among the top three features to predict if a post-release reopened bug will get resolved rapidly/slowly. It signifies that features related to developer comments are important in predicting if a release based reopened bug will get resolved rapidly/slowly. Developers post comments in bug reports regarding their progress of bug resolution, challenges they are facing to resolve the bug, and other useful discussions related to the bug. For example,

in a post-release reopened bug report [13] (id: 8786) for the Apache Yarn project, the bug is reopened for almost two years and still no satisfactory fix is provided and a developer comments *"Jason Darrell Lowe Eric Badger Jon Bender do we still need this JIRA open? Is the issue still happening after YARN-9833? (as it turned out that fix is still not 100% perfect, but very close enough to 100% which makes it acceptable)."* We recommend the research community to perform a study (either qualitative study or model based analysis) of developer comments and determine the reasons which lead to a very high rework time for pre and post-release reopened bugs. This study can help in the understanding of in-depth issues that lead to an increase in release based reopened bugs rework time.

4.5 Threats

4.5.1 External Validity

In this chapter, we investigate whether we can predict if a release based reopened bug will get resolved rapidly/slowly. We considered JIRA bug tracking systems [9] in our study because of their popularity in the developer community. However, our findings may not be generalizable for release based reopened bugs that are tracked by other systems (such as Bugzilla bug tracking systems). In bug tracking systems (such as JIRA) some projects don't follow the practice of tagging an affected version to a bug report. In those projects, very few or none of the bugs have tagged affected version. For our study, we classify such bugs as unidentified release bugs and ignore these bugs from

[13] https://issues.apache.org/jira/browse/YARN-8786

our study. However, these unidentified release bugs might be related to an affected version. We don't have a mechanism to determine if a bug is related to an affected version when the *affected version* field of the bug report is not filled by developers. We have used all JIRA projects for our study irrespective of how good/bad they follow the practice of tagging an affected version in the bug report. We encourage future studies to select projects based on criteria such as the percentage of bugs with a tagged affected version and revisit our study to determine if the results of our prediction pipelines change significantly.

4.5.2 Internal Validity

For our qualitative study, we consider 100 bugs from each pre-release and post-release reopened bugs. We do not claim to find all the reasons to reopen bugs in pre-release and post-release reopened bugs. We encourage future studies to investigate a larger sample of reopened bugs for each category. We only study top 20% and bottom 20% bugs as fast resolved and slow resolved respectively based on the rework time. We follow this approach to ensure that there is a sharp distinction between bugs that are fast closed and bugs that are slow closed. This approach is motivated by prior studies on selecting distinct datasets [54, 93]. Changing the 20% threshold values can change the performance of prediction pipelines. However, we have chosen the threshold value from a prior study [54]. We choose 7 learners (i.e., Random Forest, Logistic Regression, Gradient Boosting, Adaboost, Decision Tree, Multinomial Naive Bayes, and KNN) to predict if a release based reopened bug will get resolved rapidly/slowly. Prior studies show that different learners can have different prediction performances on a given

dataset [29, 73]. However, the selection of these 7 learners is inspired by the study conducted by Agrawal et al. [1]. Agrawal et al. considered 5 learners (i.e., Random Forest, Logistic Regression, Decision Tree, Multinomial Naive Bayes, and KNN) for their study. Apart from the above 5 learners, we also use two more learners (i.e., Adaboost, and Gradient Boosting) because they are widely used in prior prediction studies [29, 90, 91].

We consider the AUC measure to determine the performance of our prediction pipelines whereas other performance metrics such as precision, recall, and F1-score can also be used to evaluate the performance of our prediction pipelines. We do so, as several prior studies advocate using AUC performance measure to determine the performance of prediction pipelines in software analytics [30, 55, 73, 74, 91]. We encourage future researchers to replicate our study by using other performance measures (such as precision, recall, and F1-score) and compare the order of performance of different studied learners with our observed order of those learners using the AUC performance measure. This can help determine the overall performance of learners across multiple performance measures.

4.5.3 Construct Validity

In our study, we classify learners that give an AUC above 0.7 as learners with an acceptable performance in determining if a release based reopened bug will get resolved rapidly/slowly. We base our model's feature importance analysis based on learners with acceptable performance. Changing the learner AUC performance threshold value of 0.7 will lead to adding/removing the learners for feature importance analysis. This will change the results of feature importance analysis. However, we observe that many prior studies use 0.7 as a threshold value of AUC to be classified as acceptable [3, 45,

66, 67, 97].

We use a grid search based hyperparameters tuning for our prediction pipelines. Our decision is motivated by the work of Tantithamthavorn et al. [90]. Tantithamthavorn et al. discuss that different hyperparameters tuning methods (such as random search, grid search, and genetic algorithm based methods) yield similar performance results.

4.6 Chapter Summary

Reopened bugs involve a lot of rework effort of software development teams. Reopened bugs can be categorized into two categories based on software-release, these are pre-release reopened bugs and post-release reopened bugs. We conduct a study of 23,350 reopened bugs from 462 projects in JIRA. We tag reopened bugs as pre-release reopened and post-release reopened bugs based on the release date of affected versions of the bugs. We share the findings of our study: 1) We provide a novel way to classify a reopened bug as pre-release and post-release by tagging a unique bug {bug id, release version, reopened count} as either pre-release or post-release reopened bug based on the release of its affected version. 2) Pre-release reopened bugs take 57.7%, 84.6%, and 87.6% less rework time than post-release reopened bugs when the bug is reopened due to patch, integration and incorrect assessment issues respectively. 3) At best we are able to predict whether a post-release reopened bug will get resolved rapidly/slowly with an acceptable AUC of 0.78 which is 6% greater than for the same prediction pipeline using pre-release reopened bugs. Based on the findings of our study, we recommend researchers segregate post-release reopened bugs from pre-release reopened bugs and use post-release reopened bugs for the reopened bugs study since

post-release reopened bugs are more critical to resolve. Future studies can leverage features from our study to get a better understanding of post-release reopened bugs, based on the fact that these features are important to correctly predict the resolution speed of a post-release reopened bug.

CHAPTER 5

Conclusions and Future Work

THIS chapter summarizes our work and presents potential opportunities for the future work.

A reopened bug can take considerably more time to fix, and developers typically spend a lot of effort in fixing reopened bugs. Reopened bugs also lead to a loss of end-users trust in the quality of the software. Post-release reopened bugs are crucial to resolving on time as they involve a larger audience (i.e., clients).

In this thesis, we revisit reopened bug prediction on a large dataset consisting of 47 projects tracked by JIRA and using an updated pipeline for reopened bug prediction. Using our updated pipeline, we observe that only 34% of our studied 47 projects have

an acceptable performance (i.e., AUC $\geqslant$ 0.7) in determining the likelihood of reopening a bug. We also segregate reopened bugs based on software release, these are pre-release reopened bugs and post-release reopened bugs. We observe that pre-release reopened bugs take 57.7%, 84.6%, and 87.6% less rework time than post-release reopened bugs when the bug is reopened due to patch, integration, and incorrect assessment issues respectively. Moreover, we observe that at best we are able to predict whether a post-release reopened bug will get resolved rapidly/slowly with an acceptable AUC of 0.78. We share our thesis contribution.

5.1 Thesis Contributions

The main contributions of our thesis are listed below.

1. Large scale study of reopened bug characteristics with two case studies of reopened bug prediction and fixing speed prediction for post-release reopened bugs.

2. We improve the prior reopened bug prediction pipeline and demonstrate that only 34% projects give an acceptable AUC ($\geqslant$ 0.7) to predict reopened bugs.

3. We find that for pre-release reopened bugs the median rework time is twice as much as post-release reopened bugs indicating that pre-release and post-release reopened bugs have different characteristics.

4. We observe that post-release reopened bugs give an AUC of 0.78 for predicting if a release based reopened bug will get resolved rapidly, which is 6% greater than AUC obtained for the same model using pre-release reopened bugs. Therefore,

studying reopened bugs as a whole leads to biased implications. We encourage researchers to segregate pre-release and post-release reopened bugs in their analysis.

5.2 Future work

In this section, we discuss potential opportunities for improving our study.

- We encourage bug tracking systems to introduce tags for documentation issues (i.e., bookkeeping). These documentation issues, are a noise as they do not contain any technical details about bug reopening activity. By using these tags, studies on prediction of reopened bugs can avoid bugs due to documentation issues for the prediction task.

- We also encourage future research to leverage the rich data available during and after a bug is reopened, while this data was not previously used in understanding reopened bugs. We further encourage researchers to explore the management of reopened bugs based on the whole bug history to understand their eventual resolution.

Bibliography

[1] Agrawal, A., Fu, W., Chen, D., Shen, X., and Menzies, T. (2019). How to" dodge" complex software analytics. *IEEE Transactions on Software Engineering (TSE'19)*, pages 1–13.

[2] Agrawal, A. and Menzies, T. (2018). Is" better data" better than" better data miners"? In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE'18)*, pages 1050–1061. IEEE.

[3] Al Dallal, J. and Morasca, S. (2014). Predicting object-oriented class reuse-proneness using internal quality attributes. *Empirical Software Engineering (EMSE'14)*, 19(4):775–821.

[4] Alkharusi, H. (2012). Categorical variables in regression analysis: A comparison of dummy and effect coding. *International Journal of Education*, 4(2):202.

[5] Altmann, A., Toloşi, L., Sander, O., and Lengauer, T. (2010). Permutation importance: a corrected feature importance measure. *Bioinformatics*, 26(10):1340–1347.

[6] Anvik, J., Hiew, L., and Murphy, G. C. (2006). Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering (ICSE'06)*, pages 361–370.

[7] Arellano, A. V. (2019). *Epidemiological Disease Surveillance Using Public Media Text Mining*. North Carolina State University.

[8] Artstein, R. and Poesio, M. (2008). Inter-coder agreement for computational linguistics. *Computational Linguistics*, 34(4):555–596.

[9] Atlassian (2020). The #1 software development tool used by agile teams https://www.atlassian.com/software/jira, [Online; accessed 02-June-2020].

[10] Beckler, D. T., Thumser, Z. C., Schofield, J. S., and Marasco, P. D. (2018). Reliability in evaluator-based tests: using simulation-constructed models to determine contextually relevant agreement thresholds. *BMC medical research methodology*, 18(1):1–12.

[11] Bell, R. M., Ostrand, T. J., and Weyuker, E. J. (2006). Looking for bugs in all the right places. In *Proceedings of the 2006 international symposium on Software testing and analysis*, pages 61–72.

[12] Biggers, L. R., Bocovich, C., Capshaw, R., Eddy, B. P., Etzkorn, L. H., and Kraft, N. A. (2014). Configuring latent dirichlet allocation based feature location. *Empirical Software Engineering (EMSE'14)*, 19(3):465–500.

[13] Bortis, G. and Van Der Hoek, A. (2013). Porchlight: A tag-based approach to bug triaging. In *2013 35th International Conference on Software Engineering (ICSE'13)*, pages 342–351. IEEE.

[14] Boslaugh, S. (2012). *Statistics in a nutshell: A desktop quick reference*. O'Reilly Media, Inc.

[15] Caglayan, B., Misirli, A. T., Miranskyy, A., Turhan, B., and Bener, A. (2012). Factors characterizing reopened issues: a case study. In *Proceedings of the 8th International Conference on Predictive Models in Software Engineering*, pages 1–10.

[16] Cantor, A. B. (1996). Sample-size calculations for cohen's kappa. *Psychological methods*, 1(2):150.

[17] Cerda, P., Varoquaux, G., and Kégl, B. (2018). Similarity encoding for learning with dirty categorical variables. *Machine Learning*, 107(8-10):1477–1494.

[18] Chawla, N. V., Bowyer, K. W., Hall, L. O., and Kegelmeyer, W. P. (2002). SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357.

[19] Chen, D., Fu, W., Krishna, R., and Menzies, T. (2018). Applications of psychological science for actionable analytics. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18)*, pages 456–467.

[20] Chen, T.-H., Nagappan, M., Shihab, E., and Hassan, A. E. (2014). An empirical study of dormant bugs. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR'14)*, pages 82–91.

[21] Corazza, A., Di Martino, S., Maggio, V., and Scanniello, G. (2016). Weighing lexical information for software clustering in the context of architecture recovery. *Empirical Software Engineering (EMSE'16)*, 21(1):72–103.

[22] Corbin, J. M. and Strauss, A. (1990). Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative sociology*, 13(1):3–21.

[23] Da Costa, D. A., McIntosh, S., Shang, W., Kulesza, U., Coelho, R., and Hassan, A. E. (2016). A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering (TSE'16)*, 43(7):641–657.

[24] da Costa, D. A., McIntosh, S., Treude, C., Kulesza, U., and Hassan, A. E. (2018). The impact of rapid release cycles on the integration delay of fixed issues. *Empirical Software Engineering (EMSE'18)*, 23(2):835–904.

[25] Denny, M. and Spirling, A. (2017). Text preprocessing for unsupervised learning: Why it matters, when it misleads, and what to do about it. *When It Misleads, and What to Do about It (September 27, 2017)*.

[26] Erlikh, L. (2000). Leveraging legacy system dollars for e-business. *IT professional*, 2(3):17–23.

[27] Fu, W., Menzies, T., and Shen, X. (2016). Tuning for software analytics: Is it really necessary? *Information and Software Technology (IST'16)*, 76:135–146.

[28] Gehan, E. A. (1965). A generalized wilcoxon test for comparing arbitrarily singly-censored samples. *Biometrika*, 52(1-2):203–224.

[29] Ghotra, B., McIntosh, S., and Hassan, A. E. (2015). Revisiting the impact of classification techniques on the performance of defect prediction models. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE'15)*, volume 1, pages 789–800. IEEE.

[30] Ghotra, B., McIntosh, S., and Hassan, A. E. (2017). A large-scale study of the impact of feature selection techniques on defect classification models. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR'17)*, pages 146–157. IEEE.

[31] Giger, E., Pinzger, M., and Gall, H. (2010). Predicting the fix time of bugs. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, pages 52–56.

[32] Guo, P. J., Zimmermann, T., Nagappan, N., and Murphy, B. (2010). Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)-Volume 1*, pages 495–504.

[33] Guo, P. J., Zimmermann, T., Nagappan, N., and Murphy, B. (2011). " not my bug!" and other reasons for software bug report reassignments. In *Proceedings of the ACM 2011 conference on Computer supported cooperative work*, pages 395–404.

[34] Hardeniya, N., Perkins, J., Chopra, D., Joshi, N., and Mathur, I. (2016). *Natural language processing: python and NLTK*. Packt Publishing Ltd.

[35] Hayes, A. F. and Krippendorff, K. (2007). Answering the call for a standard reliability measure for coding data. *Communication methods and measures*, 1(1):77–89.

[36] He, H. and Ma, Y. (2013). *Imbalanced learning: foundations, algorithms, and applications*. John Wiley & Sons.

[37] Hébert, A. (2020). *Estimation of Road Accident Risk with Machine Learning*. PhD thesis, Concordia University.

[38] Hemalatha, I., Varma, G. S., and Govardhan, A. (2012). Preprocessing the informal text for efficient sentiment analysis. *International Journal of Emerging Trends & Technology in Computer Science (IJETTCS'12)*, 1(2):58–61.

[39] Herzig, K. (2014). Using pre-release test failures to build early post-release defect prediction models. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 300–311. IEEE.

[40] Jalbert, N. and Weimer, W. (2008). Automated duplicate detection for bug tracking systems. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN'08)*, pages 52–61. IEEE.

[41] Jiang, Z., Sarkar, S., and Jacob, V. S. (2012). Postrelease testing and software release policy for enterprise-level systems. *Information Systems Research*, 23(3-part-1):635–657.

[42] Jiarpakdee, J., Tantithamthavorn, C., and Hassan, A. E. (2018a). The impact of correlated metrics on defect models. *arXiv preprint arXiv:1801.10271*.

[43] Jiarpakdee, J., Tantithamthavorn, C., and Hassan, A. E. (2019). The impact of correlated metrics on the interpretation of defect models. *IEEE Transactions on Software Engineering (TSE'19)*.

[44] Jiarpakdee, J., Tantithamthavorn, C., and Treude, C. (2018b). Autospearman: Automatically mitigating correlated metrics for interpreting defect models. *arXiv preprint arXiv:1806.09791*.

[45] Jiarpakdee, J., Tantithamthavorn, C., and Treude, C. (2020). The impact of automated feature selection techniques on the interpretation of defect models. *Empirical Software Engineering (EMSE'20)*, 25(5):3590–3638.

[46] JIRA (2019). What are issue statuses, priorities, and resolutions? https://support.atlassian.com/jira-cloud-administration/docs/what-are-issue-statuses-priorities-and-resolutions/, [Online; accessed 2-July-2019].

[47] Kannan, S. and Gurusamy, V. (2014). Preprocessing techniques for text mining. *International Journal of Computer Science & Communication Networks*, 5(1):7–16.

[48] Kaufman, S., Rosset, S., Perlich, C., and Stitelman, O. (2012). Leakage in data mining: Formulation, detection, and avoidance. *ACM Transactions on Knowledge Discovery from Data (TKDD'12)*, 6(4):1–21.

[49] Khomh, F., Adams, B., Dhaliwal, T., and Zou, Y. (2015). Understanding the impact of rapid releases on software quality. *Empirical Software Engineering (EMSE'15)*, 20(2):336–373.

[50] Khomh, F., Dhaliwal, T., Zou, Y., and Adams, B. (2012). Do faster releases improve software quality? an empirical case study of mozilla firefox. In *2012 9th IEEE working conference on mining software repositories (MSR'12)*, pages 179–188. IEEE.

[51] Kikas, R., Dumas, M., and Pfahl, D. (2016). Using dynamic and contextual features

to predict issue lifetime in github projects. In *2016 ieee/acm 13th working conference on mining software repositories (MSR'16)*, pages 291–302. IEEE.

[52] Kochhar, P. S., Lo, D., Lawall, J., and Nagappan, N. (2017). Code coverage and postrelease defects: A large-scale study on open source projects. *IEEE Transactions on Reliability*, 66(4):1213–1228.

[53] Kvålseth, T. O. (1989). Note on cohen's kappa. *Psychological reports*, 65(1):223–226.

[54] Lee, D., Rajbahadur, G. K., Lin, D., Sayagh, M., Bezemer, C.-P., and Hassan, A. E. (2020). An empirical study of the characteristics of popular minecraft mods. *Empirical Software Engineering (EMSE'20)*, 25(5):3396–3429.

[55] Lessmann, S., Baesens, B., Mues, C., and Pietsch, S. (2008). Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering (TSE'08)*, 34(4):485–496.

[56] Li, H., Shang, W., Adams, B., Sayagh, M., and Hassan, A. E. (2020). A qualitative study of the benefits and costs of logging from developers' perspectives. *IEEE Transactions on Software Engineering (TSE'20)*.

[57] Lipton, Z. C. (2018). The mythos of model interpretability. *Queue*, 16(3):31–57.

[58] Malhotra, R. and Khanna, M. (2017). An empirical study for software change prediction using imbalanced data. *Empirical Software Engineering (EMSE'17)*, 22(6):2806–2851.

[59] Marks, L., Zou, Y., and Hassan, A. E. (2011). Studying the fix-time for bugs in large open source projects. In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, pages 1–8.

[60] McIntosh, S., Kamei, Y., Adams, B., and Hassan, A. E. (2016). An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering (EMSE'16)*, 21(5):2146–2189.

[61] McMillan, C., Grechanik, M., Poshyvanyk, D., Fu, C., and Xie, Q. (2011). Exemplar: A source code search engine for finding highly relevant applications. *IEEE Transactions on Software Engineering (TSE'11)*, 38(5):1069–1087.

[62] Méndez, J. R., Iglesias, E. L., Fdez-Riverola, F., Díaz, F., and Corchado, J. M. (2005). Tokenising, stemming and stopword removal on anti-spam filtering domain. In *Conference of the Spanish Association for Artificial Intelligence*, pages 449–458. Springer.

[63] Meyer, T. A. and Whateley, B. (2004). Spambayes: Effective open-source, bayesian based, email classification system. In *CEAS*. Citeseer.

[64] Mi, Q. and Keung, J. (2016). An empirical analysis of reopened bugs based on open source projects. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, pages 1–10.

[65] Mi, Q., Keung, J., Huo, Y., and Mensah, S. (2018). Not all bug reopens are negative: A case study on eclipse bug reports. *Information and Software Technology (IST'18)*, 99:93–97.

[66] Morasca, S. and Lavazza, L. (2020). On the assessment of software defect prediction models via ROC curves. *Empirical Software Engineering (EMSE'20)*, 25(5):3977–4019.

[67] Muller, M. P., Tomlinson, G., Marrie, T. J., Tang, P., McGeer, A., Low, D. E., Detsky, A. S., and Gold, W. L. (2005). Can routine laboratory tests discriminate between severe acute respiratory syndrome and other causes of community-acquired pneumonia? *Clinical infectious diseases*, 40(8):1079–1086.

[68] Munaiah, N., Camilo, F., Wigham, W., Meneely, A., and Nagappan, M. (2017). Do bugs foreshadow vulnerabilities? an in-depth study of the chromium project. *Empirical Software Engineering (EMSE'17)*, 22(3):1305–1347.

[69] Murphy, G. and Cubranic, D. (2004). Automatic bug triage using text categorization. In *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*, pages 1–6. Citeseer.

[70] NLTK (2021). Natural Language Toolkit https://www.nltk.org/, [Online; accessed 20-Apr-2021].

[71] Nyamawe, A. S., Liu, H., Niu, N., Umer, Q., and Niu, Z. (2020). Feature requests-based recommendation of software refactorings. *Empirical Software Engineering (EMSE'20)*, 25(5):4315–4347.

[72] Panjer, L. D. (2007). Predicting eclipse bug lifetimes. In *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*, pages 29–29. IEEE.

[73] Rajbahadur, G. K., Wang, S., Kamei, Y., and Hassan, A. E. (2017). The impact of using regression models to build defect classifiers. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR'17)*, pages 135–145. IEEE.

[74] Rajbahadur, G. K., Wang, S., Kamei, Y., and Hassan, A. E. (2019). Impact of discretization noise of the dependent variable on machine learning classifiers in software engineering. *IEEE Transactions on Software Engineering (TSE'19)*, pages 1–18.

[75] Rakha, M. S., Bezemer, C.-P., and Hassan, A. E. (2017). Revisiting the performance evaluation of automated approaches for the retrieval of duplicate issue reports. *IEEE Transactions on Software Engineering (TSE'17)*, 44(12):1245–1268.

[76] Rees-Jones, M., Martin, M., and Menzies, T. (2017). Better predictors for issue lifetime. *arXiv preprint arXiv:1702.07735*.

[77] Regenschein (2016). Naive Bayes questions: continus data, negative data, and MultinomialNB in scikit-learn. https://stats.stackexchange.com/questions/169400/naive-bayes-questions-continus-data-negative-data-and-multinomialnb-in-scikit, [Online; accessed 08-June-2016].

[78] Rwemalika, R., Kintis, M., Papadakis, M., Le Traon, Y., and Lorrach, P. (2019). An industrial study on the differences between pre-release and post-release bugs. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME'19)*, pages 92–102. IEEE.

[79] Saha, R. K., Khurshid, S., and Perry, D. E. (2015). Understanding the triaging and fixing processes of long lived bugs. *Information and software technology (IST'15)*, 65:114–128.

[80] Schröter, A., Zimmermann, T., Premraj, R., and Zeller, A. (2006). If your bug

database could talk. In *Proceedings of the 5th international symposium on empirical software engineering*, volume 2, pages 18–20. Citeseer.

[81] Scikit-learn (2020). Tuning the hyper-parameters of an estimator. https://scikit-learn.org/stable/modules/grid$_search.html$grid−search,[Online; accessed 08-June-2020].

[82] Scoccia, G. L. and Autili, M. (2020). Web frameworks for desktop apps: an exploratory study. In *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'20)*, pages 1–6.

[83] Shihab, E., Ihara, A., Kamei, Y., Ibrahim, W. M., Ohira, M., Adams, B., Hassan, A. E., and Matsumoto, K.-i. (2010). Predicting re-opened bugs: A case study on the eclipse project. In *2010 17th Working Conference on Reverse Engineering (WCRE'10)*, pages 249–258. IEEE.

[84] Shihab, E., Ihara, A., Kamei, Y., Ibrahim, W. M., Ohira, M., Adams, B., Hassan, A. E., and Matsumoto, K.-i. (2013). Studying re-opened bugs in open source software. *Empirical Software Engineering (EMSE'13)*, 18(5):1005–1042.

[85] Soldano, A. (2014). duplicate WSDL elements generated. https://issues.apache.org/jira/browse/CXF-6089?, [Online; accessed 11-November-2014].

[86] Somasundaram, K. and Murphy, G. C. (2012). Automatic categorization of bug reports using latent dirichlet allocation. In *Proceedings of the 5th India software engineering conference*, pages 125–130.

[87] Song, F., Liu, S., and Yang, J. (2005). A comparative study on text representation schemes in text categorization. *Pattern analysis and applications*, 8(1-2):199–209.

[88] Srividhya, V. and Anitha, R. (2010). Evaluating preprocessing techniques in text categorization. *International journal of computer science and application*, 47(11):49–51.

[89] Stolberg, S. (2009). Enabling agile testing through continuous integration. In *2009 agile conference*, pages 369–374. IEEE.

[90] Tantithamthavorn, C., Hassan, A. E., and Matsumoto, K. (2018). The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. *IEEE Transactions on Software Engineering (TSE'18)*.

[91] Tantithamthavorn, C., McIntosh, S., Hassan, A. E., and Matsumoto, K. (2016a). Automated parameter optimization of classification techniques for defect prediction models. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*, pages 321–332.

[92] Tantithamthavorn, C., McIntosh, S., Hassan, A. E., and Matsumoto, K. (2016b). An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering (TSE'16)*, 43(1):1–18.

[93] Tian, Y., Nagappan, M., Lo, D., and Hassan, A. E. (2015). What are the characteristics of high-rated apps? a case study on free android applications. In *2015 IEEE international conference on software maintenance and evolution (ICSME'15)*, pages 301–310. IEEE.

[94] Tian, Y., Sun, C., and Lo, D. (2012). Improved duplicate bug report identification. In *2012 16th European Conference on Software Maintenance and Reengineering (CSMR'12)*, pages 385–390. IEEE.

[95] Tian, Y., Wijedasa, D., Lo, D., and Le Goues, C. (2016). Learning to rank for bug report assignee recommendation. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC'16)*, pages 1–10. IEEE.

[96] Tu, F., Zhu, J., Zheng, Q., and Zhou, M. (2018). 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 (ESEC/FSE '18)*, pages 307–318.

[97] Turabieh, H., Mafarja, M., and Li, X. (2019). Iterated feature selection algorithms with layered recurrent neural network for software fault prediction. *Expert systems with applications*, 122:27–42.

[98] Uysal, A. K. and Gunal, S. (2014). The impact of preprocessing on text classification. *Information Processing & Management*, 50(1):104–112.

[99] Vassallo, C., Panichella, S., Palomba, F., Proksch, S., Gall, H. C., and Zaidman, A. (2020). How developers engage with static analysis tools in different contexts. *Empirical Software Engineering (EMSE'20)*, 25(2):1419–1457.

[100] Verleysen, M. and François, D. (2005). The curse of dimensionality in data mining and time series prediction. In *International work-conference on artificial neural networks*, pages 758–770. Springer.

[101] Vieira, R., da Silva, A., Rocha, L., and Gomes, J. P. (2019). From reports to bug-fix commits: A 10 years dataset of bug-fixing activity from 55 apache's open source projects. In *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 80–89.

[102] Viera, A. J., Garrett, J. M., et al. (2005). Understanding interobserver agreement: the kappa statistic. *Fam med*, 37(5):360–363.

[103] Webb, J. K., Keller, K. A., Welle, K., and Allender, M. C. (2020). Evaluation of the inter-and intraindividual agreement of a pododermatitis scoring model in greater flamingos (phoenicopterus roseus). *Journal of Zoo and Wildlife Medicine*, 51(2):379–384.

[104] Xia, X., Lo, D., Ding, Y., Al-Kofahi, J. M., Nguyen, T. N., and Wang, X. (2016). Improving automated bug triaging with specialized topic model. *IEEE Transactions on Software Engineering (TSE'16)*, 43(3):272–297.

[105] Xia, X., Lo, D., Shihab, E., Wang, X., and Zhou, B. (2015). Automatic, high accuracy prediction of reopened bugs. *Automated Software Engineering (ASE'15)*, 22(1):75–109.

[106] Xia, X., Lo, D., Wang, X., Yang, X., Li, S., and Sun, J. (2013a). A comparative study of supervised learning algorithms for re-opened bug prediction. In *2013 17th European Conference on Software Maintenance and Reengineering (CSMR'13)*, pages 331–334. IEEE.

[107] Xia, X., Lo, D., Wang, X., and Zhou, B. (2013b). Accurate developer recommendation for bug resolution. In *2013 20th Working Conference on Reverse Engineering (WCRE'13)*, pages 72–81. IEEE.

[108] Xia, X., Lo, D., Wen, M., Shihab, E., and Zhou, B. (2014). An empirical study of bug report field reassignment. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE'14)*, pages 174–183. IEEE.

[109] Xuan, J., Jiang, H., Hu, Y., Ren, Z., Zou, W., Luo, Z., and Wu, X. (2014). Towards effective bug triage with software data reduction techniques. *IEEE transactions on knowledge and data engineering*, 27(1):264–280.

[110] Xuan, J., Jiang, H., Ren, Z., and Zou, W. (2012). Developer prioritization in bug repositories. In *2012 34th International Conference on Software Engineering (ICSE'12)*, pages 25–35. IEEE.

[111] Yadav, A., Singh, S. K., and Suri, J. S. (2019). Ranking of software developers based on expertise score for bug triaging. *Information and Software Technology (IST'19)*, 112:1–17.

[112] Yao, G. (2016). Last action time should be initialized as created time instead of 0. https://issues.apache.org/jira/browse/HAWQ-737?, [Online; accessed 14-July-2016].

[113] Yatish, S., Jiarpakdee, J., Thongtanunam, P., and Tantithamthavorn, C. (2019). Mining software defects: should we consider affected releases? In *IEEE/ACM 41st International Conference on Software Engineering (ICSE'19)*, pages 654–665. IEEE.

[114] Yu, X., Li, J., and Zhong, H. (2009). on reducing the pre-release failures of web plug-in on social networking site. In *International Conference on Software Process (ICSP'09)*, pages 236–245. Springer.

[115] Zaman, S., Adams, B., and Hassan, A. E. (2012). A qualitative study on performance bugs. In *2012 9th IEEE working conference on mining software repositories (MSR'12)*, pages 199–208. IEEE.

[116] Zeng, Y., Jiang, K., and Chen, J. (2019). Automatic seismic salt interpretation with deep convolutional neural networks. In *Proceedings of the 3rd International Conference on Information System and Data Mining (ICISDM'19)*, pages 16–20.

[117] Zhang, H., Wang, S., Chen, T.-H., and Hassan, A. E. (2020). Are comments on stack overflow well organized for easy retrieval by developers? *ACM Transactions on Software Engineering and Methodology (TOSEM'20)*, 29.

[118] Zhang, H., Wang, S., Chen, T.-H., Zou, Y., and Hassan, A. E. (2019). An empirical study of obsolete answers on Stack Overflow. *IEEE Transactions on Software Engineering (TSE'19)*.

[119] Zheng, A. and Casari, A. (2018). *Feature engineering for machine learning: principles and techniques for data scientists*. " O'Reilly Media, Inc.".

[120] Zimmermann, T., Nagappan, N., Guo, P. J., and Murphy, B. (2012). Characterizing and predicting which bugs get reopened. In *2012 34th International Conference on Software Engineering (ICSE'12)*, pages 1074–1083. IEEE.

[121] Zimmermann, T., Nagappan, N., and Zeller, A. (2008). Predicting bugs from history. In *Software evolution*, pages 69–88. Springer.