

Release-Aware and Prioritized Bug-Fixing Task Assignment Optimization

March 2020

Graduate School of Systems Engineering

Wakayama University

Yutaro Kashiwa

リリースと優先順位を考慮した
不具合修正タスク割当の最適化

令和 2 年 3 月

和歌山大学大学院システム工学研究科

柏祐太郎

Abstract

Software plays a crucial role in gaining a competitive advantage in the market. Software should meet user needs and be rapidly delivered. Therefore, modern software development projects are forced to shorten the intervals between releases. However, the size of the products and the scale of product development are growing due to increasing demands for multiple system functionalities or integrated systems. Increasing the scale of a product leads to an increase in the number of bugs in the product. Developers are in a dilemma to fix numerous bugs within a short development period, which is shrinking over the years.

Developers cannot fix all the reported bugs by the next release date; therefore, managers are required to prioritize the bugs to be fixed before the release and assign developers to fix those bugs. The managers utilize a significant amount of time for this work because of the large number of bug reports that have to be read.

To mitigate the workload, many bug assignment methods have been proposed with an aim to automate the assignments. However, Park et al. report that most of the methods tend to concentrate the assignment of bugs to specific developers. The task of concentrating on specific developers, by the traditional methods, would reduce the number of bugs that they can actually fix by the next release date. This is because most projects have releases planned in advance, and the scope of the developers (even well-experienced) is limited to the number of bug-fixes by the release date.

This thesis proposes the Release-Aware and Prioritized Task-assignment Optimization Framework (RAPTOR), which moderates the bug-fixing loads for specific developers in order to increase the number of bugs fixed by the release date. In this thesis, we show that RAPTOR: (1) can mitigate the situation in which bug-fixing tasks are concentrated to a small number of developers; (2) increases the number of high priority bugs that developers can fix by the next release date; (3) reduces the time developers devote to fixing bugs, compared with the manual bug triaging method and other existing methods.

Keywords

Bug triage, Project management, Quality assurance, Optimization

概要

ソフトウェアはビジネスにおける優位性を獲得する上で重要な役割を担っている。特に近年ではユーザーの獲得競争が激しいため、ユーザーが求める機能や品質を持つソフトウェアを、より早くユーザーに届けることが重要である。そのため、近年のソフトウェア開発では、納期の間隔を短くする対応が取られている。一方、さまざまな機能や他のシステムとの連携に対する需要の高まりにより、製品と開発が大規模化し、ソフトウェアに発生する不具合数も比例して増加している。そのため、開発者は年々短縮化している開発期間で、多くの不具合を修正しなければならない状況に置かれている。

しかし、日々報告される不具合数が膨大であるため、開発者はすべての不具合を修正することは難しい。そのため、プロジェクトの管理者が不具合に優先順位を付け、優先度が高い不具合を中心に、担当開発者を割当て、不具合を修正している。これらの作業では、プロジェクトの管理者が不具合報告の内容を精査する必要があり、多大な時間を消費していると言われている。

近年ではこれらの負担を軽減するために、優先度付けおよびタスク割当ての自動化を目的とした多くの手法方法が提案されている。しかし、従来方法の多くは特定の開発者に不具合を集中させる傾向があると報告されている。たとえ経験豊富な開発者であっても、リリースまでに不具合を修正できる数には限界がある。そのため、特定の開発者へのタスク集中はプロジェクト全体として、次のリリース日までに修正できる不具合の数を減少させる恐れがある。

本学位論文では、特定の開発者への負荷を緩和し、リリース日までにより多くの不具合を修正する割当手法 **Release-Aware and Prioritized Task-assignment Optimization Framework (RAPTOR)** を提案する。オープンソースソフトウェアプロジェクトの開発でデータを利用した評価実験の結果、従来手法と比べて以下の点が明らかになった。

- (1) RAPTOR は、一部の開発者へのタスク集中を緩和する
- (2) プロジェクト全体で、高優先度の不具合をリリース日までに多く修正できる
- (3) 開発者が不具合の修正に費やす時間を短縮する

Acknowledgement

This thesis would not have been possible without support from many people. I would like to thank the following for their support, wisdom, advice, encouragement, and dedication.

I sincerely appreciate my thesis committee members, Dr. Kazuhiro Kazama and Dr. Toshikazu Wada for their valuable feedback and their useful comments. Their abundant knowledge in other research areas helped to enhance my thesis.

I would like to thank my supervisor, Dr. Masao Ohira, who was also a member of my thesis committee. Since I was an undergraduate student, with little knowledge about research, he has taught me and helped me grow as a researcher. The most important thing for me is that he has shown how interesting researching is. Without him, I would not have aimed to be a researcher or grown to love researching. I would like to express my gratitude for all his guidance, support, and effort.

I would like to express my sincere gratitude to my external thesis committee member, Dr. Yasutaka Kamei. He provided keen insight into not only this thesis but also many of my publications, and also gave me a lot of opportunities to work with software engineering researchers around the world, which helped me to extend my network.

I am very fortunate to work with my collaborators and co-authors. A special note of gratitude is due to Dr. Hirohisa Aman. Inspired by his paper and his advice, I developed the idea of RAPTOR in Chapter 5. Without him, I would not have finished my thesis and would not have received any of the awards that I have been awarded over the course of my research.

I would like to thank Dr. Bram Adams. He accepted my visit to his laboratory for 1 year in total and had many meetings with me, which enhanced my study and extends my knowledge. I appreciate his effort and accumulated knowledge.

I would like to express my deepest appreciation to Dr. Akinori Ihara. Since my bachelor's, he has supported me by giving technical advice, discussing my study, and encouraging me like an older brother. I greatly appreciate his enthusiastic support.

I would like to thank the following for enhancing my study. I am extremely grateful to Dr. Naoyasu Ubayashi and Dr. Ahmed E. Hassan for inviting me to their laboratory and giving me the opportunity to discuss my research with them and their colleagues.

I would like to offer deep and sincere gratitude to Dr. Raula Gaikovina Kula and Dr. Hideaki Hata for giving me a lot of advice about not only my study but also organizing IWESEP2018, which provided me with knowledge and experience.

Special thanks to Dr. Shane McIntosh, Dr. Yuan Tian, and Dr. Mohammed Sayagh for all the fruitful discussions and giving me valuable advice on my study.

My appreciation extends to my labmates and alumni from Open Source Software Engineering Laboratory, Social Software Engineering Laboratory at Wakayama University, and Maintenance, Construction and Intelligence of Software Laboratory at Ecole Polytechnique of Montreal. They shared their valuable time with me, both during work hours and in their free time, which made my Ph.D. life enjoyable and wonderful.

Additionally, I want to thank Dr. John Michael and Anas Bouziane for giving me a lot of advice about English, which helped me to write this thesis.

I also want to thank my previous co-workers at Hitachi ltd., who encouraged me to start my Ph.D. journey. Furthermore, my Ph.D. was generously supported by the JSPS Research Fellowship for Young Scientist (DC1), the Grant-in-Aid for JSPS Fellows (JP17J03330), and Overseas Challenge Program for Young Researchers.

Finally, special thanks to my family, who have been providing me with their love and support throughout my entire life.

Contents

1	Introduction	1
1.1	Motivation and Goal	3
1.2	Thesis Overview	6
2	Background	10
2.1	Bug-Triage Using Bug Tracking Systems	10
2.2	Bug Triage and Software Release	13
2.3	Bug Prioritization	15
2.4	Bug Assignments	18
3	A Survey Study of the Bug-fixing Importance	22
3.1	Introduction	22
3.2	High Impact Bug	24
3.3	Study Design	26
3.4	Survey and Classification Results	28
3.5	Discussions	34
3.6	Chapter Summary	36
4	Release-aware Bug Priority Prediction	37
4.1	Introduction	37
4.2	Release Engineering and Bug Management	40
4.3	Study Design	45
4.4	Study Results	58
4.5	Discussion	75
4.6	Chapter Summary	81
5	Release-Aware and Prioritized Bug-fixing Task Assignments	82
5.1	Introduction	82
5.2	Multiple Knapsack Problem	84

5.3	Application of the Multiple Knapsack Problem to Bug Triage	87
5.4	Implementation	95
5.5	Experimental Design	97
5.6	Results	104
5.7	Discussions	113
5.8	Chapter Summary	118
6	Discussions	119
7	Conclusion	123

List of Figures

1.1	The overview of RAPTOR. RAPTOR prioritizes bugs and calculates the costs for bug-fixing, probabilities that each developer can fix the bug, and the workloads of each developer. With these parameters, RAPTOR generates a formula and solves it to find the best combination of bugs and developers.	5
1.2	An overview of the thesis	6
2.1	The example of bug reports in Bugzilla	11
2.2	The list of reported bugs in Bugzilla	12
2.3	The process of the bug triage affected by release pressure	14
4.1	Concept of the cycle-aware models. The cycle-aware models are trained with data in specific periods of the release cycle (e.g., the periods after releases).	43
4.2	Illustration of the procedure for creating our datasets. The data are separated by version and merged by month into 12 datasets. The developers switched to a rapid release cycle from Eclipse 4.6 after the first release (Eclipse 4.2).	47
4.3	Summary of the procedure for completing unknown metrics when new bugs are reported. The new bugs are provided through the inferred metrics by the LDA and Impute models.	52
4.4	The procedure of completing unknown metrics when new bugs are reported. The new bugs are provided with the inferred metrics by LDA and Impute models.	53
4.5	An evaluation example for the 4-splitted models. The cycle-aware models trained with quarterly training datasets predict priorities for testing datasets of the same quarter. The opaque model predicts priorities for the four quarterly testing datasets.	57
4.6	Box plot showing the number of bugs reported in each month	60

4.7	Box plot showing the number of bugs fixed in each month	60
4.8	The percentage of reported high priority bugs in each month	62
4.9	The percentage of fixed high priority bugs in each month	62
4.10	Display of percentages of the bugs reported and fixed in each month. . . .	63
4.11	Display of the percentages of the high priority bugs reported and fixed in each month.	63
4.12	An evaluation method for the 1/4/12-splitted models. The monthly models are trained with monthly training datasets, and predict priorities for testing datasets of the same month. The quarterly models and the opaque model are trained with the training datasets of every three months and all months.	69
4.13	Prediction performance of the Pure and Hybrid monthly models highlight- ing consistent superiority of the hybrid over the pure models.	71
4.14	The prediction performance of the opaque models, Hybrid quarter models and Hybrid monthly models	74
4.15	The similarity of the variables' importance in OPA and H-QuCYC	78
4.16	The similarity of the variables' importance in OPA and H-MoCYC	78
5.1	Overview of the multiple knapsack problem. The best combination, where the total value of items included in all the knapsacks is the largest, should be determined while the total weight of each knapsack is less than or equal to the capacities in all knapsacks.	85
5.2	The calculation procedure for preference. Preference is calculated by a Support Vector Machine trained with text data and the name of the fixer as a label.	89
5.3	The calculation procedure for the bug-fixing costs. The bugs are classified into categories, which indicates the median amount of bug-fixing days and thus the cost of fixing.	90
5.4	Calculation of available time slot. The available time slot T_i is calculated by subtracting the total cost assigned to D_i from the upper limit L (day) .	92

5.5	Overview of the Release-Aware and Prioritized Task-assignment Optimization fRamework (RAPTOR). RAPTOR formulates task-assignment problems with the calculated preference, costs, and available time slots.	95
5.6	Overview of our experiment. The number of fixing days are calculated by multiplying the cost with the assignment result.	103
5.7	Fixing days by each developer.	107
5.8	Relationship between the size of L and the accuracy and the number of overdue bugs	115
6.1	Datasets for RAPTOR and the cycle-aware prediction models. We use the data of three months in Eclipse 3.6 for the testing dataset. We build RAPTOR with the data for 1 year previous to the testing dataset and train the cycle-aware models with the data from Eclipse 3.0 to Eclipse 3.5. . . .	120

List of Tables

2.1	The levels of priority and severity defined on the Eclipse project web page	16
2.2	An overview of prior studies about bug-assignment methods	21
3.1	The questionnaire for our survey	27
3.2	Product domains where GitHub developers join (Q1-1)	29
3.3	Experience with OSS development (Q1-2)	29
3.4	Motivation to participate in OSS development (Q1-3)	29
3.5	A distribution of high impact bugs (Q2-1)	29
3.6	The number of categorization based on the symptoms in actual bug reports. The related researches are shown in Ref column.	31
4.1	Eclipse 3.x release dates and target reports in our dataset showing major and service releases from 2004-2011 for version 3.0-3.6.	45
4.2	The number of bugs by priority in our datasets.	46
4.3	Details of measured metrics (Daily Unit) employed in this study showing reports, closes, changes, comments, and commits.	49
4.4	Measured metrics (Bug Unit) used in this study showing reports, changes, comments, fixes, and commits.	50
4.5	Confusion matrix for a two-class problem and performance measures	58
4.6	Effect-size between quarters (+ indicates the distribution is larger than the other quarters, – indicates the distribution is smaller than others)	64
4.7	Performance of the cycle-aware models and the opaque model. The bold scores represent the highest score among the three models with statistical significance. The scores underlined indicate the highest score among the three models but without statistical significance.	67
4.8	The importance of variables in cycle-aware models and the opaque model (Importance ≥ 0.05). The bolded name of the metric indicates that the effect size is larger in RQ1 than in the other quarters.	76
5.1	List of terms used in this chapter	86

5.2	Datasets	98
5.3	Dataset contents after each filtering step	98
5.4	Statistics of fixing-time for each dataset	99
5.5	Active developers in each dataset	101
5.6	Evaluation of calculations for bug-fixing time	105
5.7	The statistics of fixing-time assigned by each method	108
5.8	Comparing the results of each method	110
5.9	The details of unfixed bugs	111
6.1	The number of unfixed bugs that the Eclipse project cannot fix by the re- lease date is shown. The table shows the results in the case that RAPTOR assign bugs with two priority prediction models (Opaque model or Cycle- aware models) or without priority predictions. The “Predicted priority” and “Actual priority” columns show the number of bugs with predicted and actual priority, respectively.	121

Chapter 1

Introduction

Software plays a crucial role in gaining a competitive advantage in the market. Software should meet user needs and be rapidly delivered. Therefore, modern software development projects are forced to shorten the intervals between releases. The importance of shortening the release intervals is represented by the popularized remark “*Release early, release often*” by Eric S. Raymond [1]. Releasing products rapidly and frequently allows developers to gain feedback from users and respond to the changing needs of users. Recently, modern software development projects, such as Google Chrome, Mozilla Firefox, and Facebook Mobile app, have released a product every 2-6 weeks. For instance, Firefox shifted from traditional intervals of 3 months to shorter intervals of 6 weeks. Firefox adopted the rapid release cycle to compete with Google Chrome’s success in the market with the same cycle.

While release cycles have become shorter, the size of the products and the scale of product development are growing due to increasing demands for multiple system functionalities or integrated systems. For instance, the size of the source code of the Google Chrome project was 1.2 million lines in the initial release, in the year 2008, but it has been constantly increasing and has reached over 25 million lines in 2019 ¹.

Increasing the scale of a product leads to an increased chance of bugs in the product.

¹OpenHub: https://www.openhub.net/p/chrome/analyses/latest/languages_summary, Last Accessed: January 2020

As bugs in a released version may result in financial losses or a loss of customers, project managers should handle the following software quality risks that induce bugs into released products:

- Residual risk

This kind of risk is where developers decide not to fix the bugs because of certain limitations, such as technical issues or time constraints. During the development, developers will try to address these risks to prevent failures in the next release. In software engineering, bug triaging studies [2, 3] mitigate this risk.

- Control risk

This kind of risk is where developers try fixing the bug, but the fix is not adequate. Therefore, the bugs will emerge again after the fix. The studies about re-opened bugs [4, 5] address this risk.

- Detection risk

This kind of risk is where a bug exists in the product but developers do not detect it. Once the bug is exposed through the testing or review process, a detection risk becomes a residual risk. The bug prediction studies [6, 7] address this risk.

- Potential risk

This kind of risk is where there are suspicious parts in source code, but they are not identified as bugs. If these parts are modified, they may turn into bugs. The technical debt studies [8, 9, 10] address this risk.

A myriad of bugs and an abundance of source code in modern software development makes project managers face difficulties in dealing with these types of risks. This thesis addresses the residual risk because project managers have been suffering from handling numerous bugs during the shrinking development periods.

1.1 Motivation and Goal

Due to the growing scale and complexity of products, projects receive a large number of bug reports. While developers have to fix numerous bugs, the amount of time they have has been shrinking over the years. Developers cannot fix all the bugs by the release date; therefore, managers are required to prioritize the bugs to be fixed before the release and assign developers to fix those bugs. Before prioritizing and assigning bugs, managers must read the bug reports. However, this work consumes a significant amount of time because of the large number of bug reports [3].

To mitigate the workload, many automated bug assignment methods have been proposed [2, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22]. The most common approach is to use classifiers such as Support Vector Machine or Naive Bayes. To assign a bug to a developer, the classifiers are trained with pairs of the description (text data) of bugs and the fixer. When a new bug is reported, the classifier takes the description of the new bug as input and assigns the relevant developers. The classifier can recommend developers who are capable of dealing with newly-reported bugs with relatively high precision (about 70%-75%).

However, these existing methods have not been utilized by projects. One of the reasons should be a task concentration problem. Park et al. reported that most of the methods tend to concentrate on assigning bugs to specific developers [23]. This is because most of the methods use machine learning classifiers, and the amount of training data for each developer is imbalanced. Developers do not fix an equal number of bugs, resulting in an imbalanced training dataset.

The tendency of the traditional methods to assign bugs to specific developers would end up reducing the number of bugs that they can fix by the next release date. This is because most projects have tight deadlines, and **even well-experienced developers are limited by time constraints**. In short and time-based release cycles adopted by modern software projects, tasks are concentrated such that products are released even though developers are aware of the numerous bugs in the products. This indicates that the existing method increases the residual risk. To reduce the residual risk, a new method

is needed to take into consideration the number of tasks that developers address and mitigate the task concentration by the existing methods [2, 18].

In addition, in order to reduce residual risk, it is necessary to decrease the impacts of the bugs contained in the product. This is because developers might not be able to fix all reported bugs by the release date, and their impacts of bugs are different. Bugs range from problems that are caused by the carelessness of the users, such as a typo in a document, to emergencies that significantly impact the users (e.g., a system crash [24])[25]. Such bugs, which have high impacts, should be specified and removed by exerting as much efforts as required by projects. Therefore, before assigning bugs to developers, we need a method to prioritize and choose the bugs that need to be fixed by the release date and bugs that can be carried over the future releases. Finally, the products that are released should have only those bugs that have a minor impact. That is, severe bugs must be fixed before the current release and minor bugs can be carried over to the next release.

In this thesis, to reduce residual risks, we construct the **R**elease-**A**ware and **P**rioritized **T**ask-assignment **O**ptimization **f**Ramework (RAPTOR). Figure 1.1 illustrates an overview of RAPTOR. RAPTOR aims to fix more number of bugs that highly impact users or developers by the release date; it also makes aware of the cost of bug-fixing and the probabilities that developers can fix bugs as well as the previous methods do. RAPTOR imposes certain limitations on assigning bugs to each developer so that the developer can fix the assigned bugs for a specific period. Under the constraints, RAPTOR can find the best combination of bugs and developers for the project so that it moderates workload and chooses bugs that severely affect users.

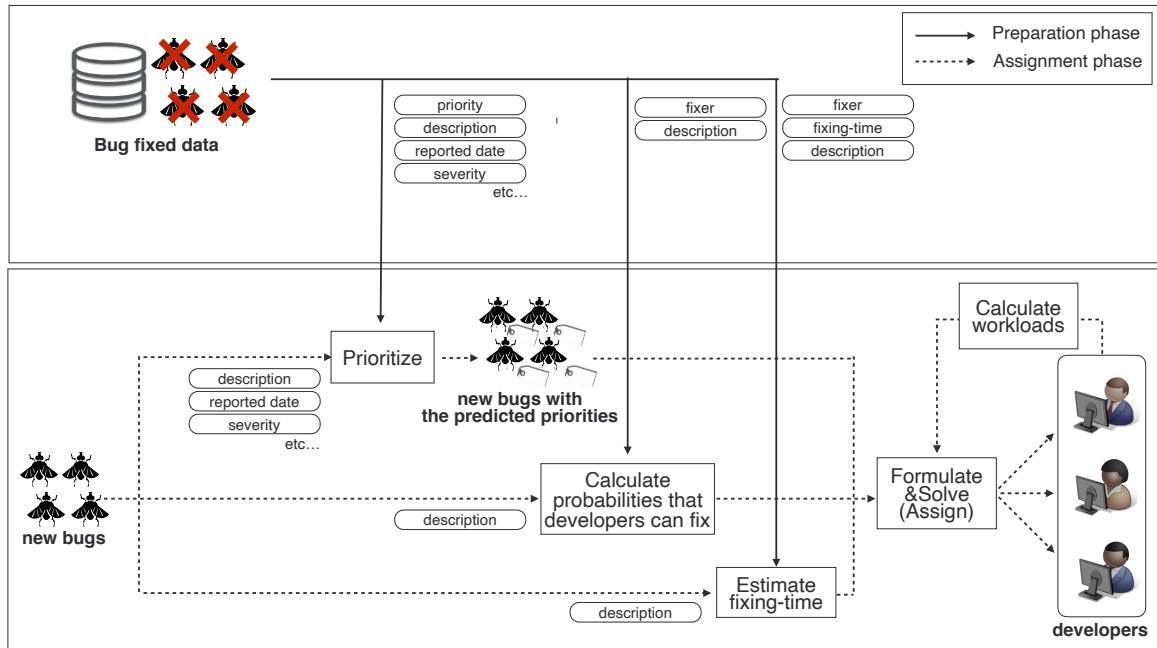


Figure 1.1: The overview of RAPTOR. RAPTOR prioritizes bugs and calculates the costs for bug-fixing, probabilities that each developer can fix the bug, and the workloads of each developer. With these parameters, RAPTOR generates a formula and solves it to find the best combination of bugs and developers.

1.2 Thesis Overview

We now provide a brief overview of this thesis (see Figure 1.2). We first provide the necessary background material about bug-triaging in Chapter 2. We conduct 3 empirical studies which are explained in Chapters 3 to 5. These studies are classified into 2 sub-goals as follows.

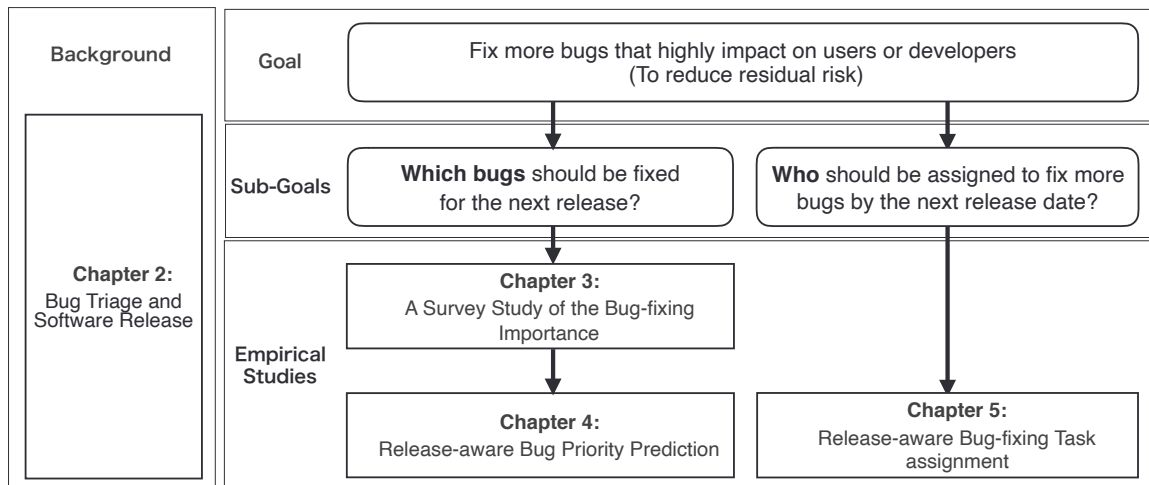


Figure 1.2: An overview of the thesis

Which bugs should be fixed for the next release?

Chapter 3: While the development period has been shrinking, the number of bugs has been raising. This makes it hard for developers to fix all reported bugs by the next release date. Therefore, developers have to decide which bugs should be fixed by the next release and which ones should be carried over to a later release. However, most of the existing literature on bug assignments do not take into account the existence of releases; therefore, they do not make decisions on the bugs that need to be fixed first, and do not consider taking decisions on carrying over some bugs to a future release. To make matters worse, most of the studies treat all bugs equally. Bugs range from problems that users do not care about (e.g., typo in documents) to emergencies that greatly concern users (e.g.,

a system crash [24])[25]. If this is not taken into consideration, it can lead to wastage of human-resource; for example, important bugs remain to be fixed by the next release.

The approaches which are similar ours are priority/severity predictions; however, they do not make decisions for the next release. In order to predict priority/severity, they use classifiers trained with various data, such as tag or text data, in bug reports. However, several studies warn that priority/severity does not indicate accurate values because 65% of bug reports are mislabeled [26]. Hence, many studies have defined serious bugs that impact on the product or process (which are named high impact bugs) [27, 28, 29, 30, 31] and built models to predict them. However, it is not sure whether developers consider these bugs as being highly impactful on the product or process. First, we address the following challenges to generate a strategy for prioritizing bugs.

Challenge I:**Identify the bugs that highly impact users or products**

In the first challenge, we interviewed 322 developers from GitHub to identify the bugs that are impactful based on those they encountered in practice. We manually inspect and classify actual bug reports from the responses. As a result, we show that there are a wide variety of high impact bugs. Particularly, developers think security and breakage bugs are highly important for FLOSS developers. Furthermore, we show that 66% of the high impact bugs have a higher importance in the projects (especially in the projects that strictly handle bugs). This helps us select bugs for the next release when the projects have a myriad of bugs.

Chapter 4: In the second challenge, we try to predict the priority, which represents the importance of bug-fixing, used by developers. The prediction will play a crucial role in selecting which bugs should be fixed by the next release date. Although some studies address building priority prediction models, the accuracy of the prediction is quite low. This is because they use data derived from the complete development process. Most projects have release cycles, and the data is produced in requirements analysis, design, implementation, test, and debug phases. The previous studies do not distinguish what developers do in each period of the release cycle.

Moreover, depending on the type of release, such as major or minor, developers switch their focus, such as either on implementing new features or fixing existing defects. Furthermore, depending on what they focus on, the characteristics of the produced data would vary. Disregarding the characteristics for specific periods makes the quality of data-driven tools lower. However, most studies about defect management support (including priority prediction work) assume that the characteristics for all periods never change across the release cycle.

Utilizing the contents of work (the characteristics) in each period would improve the accuracy. Specifically, our bug assignment method is designed for the test and debug phase, and we need a priority prediction method specialized for these periods. Therefore, we address the following challenges;

Challenge II:**Predict the importance of bugs for specific periods**

In this challenge, we show that developers' activity varies during the release cycle. Based on these findings, we build release cycle-aware models from which data is derived from appropriate periods. We conduct a case study on the Eclipse Platform project and find that cycle-aware models outperform the traditional model, which uses the bug data during the whole development.

Who should be assigned to fix more bugs by the next release date?

Chapter 5: As another important factor in decision making, we address assigning appropriate developers in order to increase bug-fixes by the next release date. This study tries to solve the problem of concentrating assignments on a small number of particular developers, which leads to a reduction in the number of bug-fixes in the projects by the next release date. Even well-experienced developers are limited to the number of bug-fixes because of the limited time before the next release. Thus, a new method is needed that takes into consideration the number of tasks that developers address and mitigates the task concentration done by the existing methods.

Challenge III:**Build the release-aware bug assignment method in order to increase bug-fixes by the next release date**

In this study, we impose limitations on assigning bugs to each developer so that the developer can fix the assigned bugs within a specific period. We formulate bug-triaging problems and the constraints that need to be imposed. We use mathematical programming to find the best combination of bugs and developers for the project.

We conduct a case study on the Eclipse Platform, GNU compiler collection, and Mozilla Firefox and show that RAPTOR (1) mitigates a situation where bug-fixing tasks are concentrated to a small number of developers; (2) increases the number of high priority bugs that developers can fix by the next release date; and (3) can reduce the time developers devote to fixing bugs, compared with the manual bug triaging method and other existing methods.

Chapter 6: Finally, we evaluate the performance of combining the above two methods, priority prediction, and task-assignment method. We find that RAPTOR with priority predictions can assign bugs so that the project can fix more bugs with higher priority by the next release date.

Chapter 2

Background


This chapter introduces how bugs in products are managed and fixed. First, we introduce bug-triage and the use of bug tracking systems. Then, we explain the relationship between software release and bug-triage. Finally, we describe the necessary activities in the bug-triage process and the related work.

2.1 Bug-Triage Using Bug Tracking Systems

When users (including developers) face a problem or find a bug, they report the bug or problem to the project. In order to receive and manage numerous reports, most projects have prepared a Bug Tracking System (abbr. BTS), such as Bugzilla, Jira, and Redmine. Figure 2.1 shows the example of bug reports in Bugzilla. The reports include various information such as the description of the bug (① in Figure 2.1) and the user's environment, such as the name of the product, version, hardware (② in Figure 2.1), and severity (③). If developers need further information to fix the bugs, the reporter discusses the issues in the form of BTS (④). Also, discussions among developers, such as about the cause or the strategy for fixing a bug, are held in the same form.

After reporters write a report and submit it, the report will be available to be seen by all developers. Managers read each of the bug reports listed in the BTSs (Figure 2.2) and confirm whether the bug needs to be fixed. If the bug needs to be fixed by the next

2.1. BUG-TRIAGE USING BUG TRACKING SYSTEMS



Bugzilla – Bug 25467Accelerator Menu performance


[Home](#) | [New](#) | [Browse](#) | [Search](#) | Search [\[?\]](#) | [Reports](#) | [Requests](#) | [Help](#) |

Bug 25467 - Accelerator Menu performance problems (GTK & Motif)


Status: RESOLVED FIXED


Alias: None

②

Product: Platform
Component: UI ([show other bugs](#))
Version: 2.1 
Hardware: PC Windows 2000

Importance: P1 critical ([vote](#))

Target Milestone: --- 

Assignee: Chris McLaren 

QA Contact:

URL:

Whiteboard:

Keywords:

Duplicates (1): [25777](#) ([view as bug list](#))


Depends on:

Blocks:


③

⑤

Attachments

The benchmark code (6.05 KB, text/plain)	no flags	Details
2002-10-30 16:04 EST , Steve Northover 		
Add an attachment (proposed patch, testcase, etc.)		View All


Note
You need to [log in](#) before you can comment on or make changes to this bug.

Chris McLaren  2002-10-28 15:24:44 EST

Description

GTK and Motif have significant performance problems using the hidden menu in its current form. I looked at a profile and found that at least 700ms is spent in AcceleratorMenu.setAccelerators on these platforms (every time we recalculate the menu - which can be as often as switching editors). I am hoping that this has to do with the fact that the hidden menu is being disposed of and recreated with new menu items each time setAccelerators is called. Can someone look at this, and perhaps reuse MenuItem instances if that proves to be the problem?


①

Steve Northover  2002-10-29 11:02:29 EST

Comment 1


This is a P1. Chris to assist SN constructing a benchmark.

④

Steve Northover  2002-10-30 16:04:57 EST

Comment 2

Created [attachment 2292](#) [\[details\]](#)
The benchmark code

Steve Northover  2002-10-30 16:36:22 EST

Comment 3

Results:
Windows 98, PII-433-256 - 110ms
Motif RedHat PII-450-256 - 438ms
GTK RedHat 8.0 PIII-500-512 - 180ms
These times don't look that bad, given the machines I am running on. Do you want me to investigate further? Is it possible that the UI code is calling this API multiple times instead of just once when switching between editors?

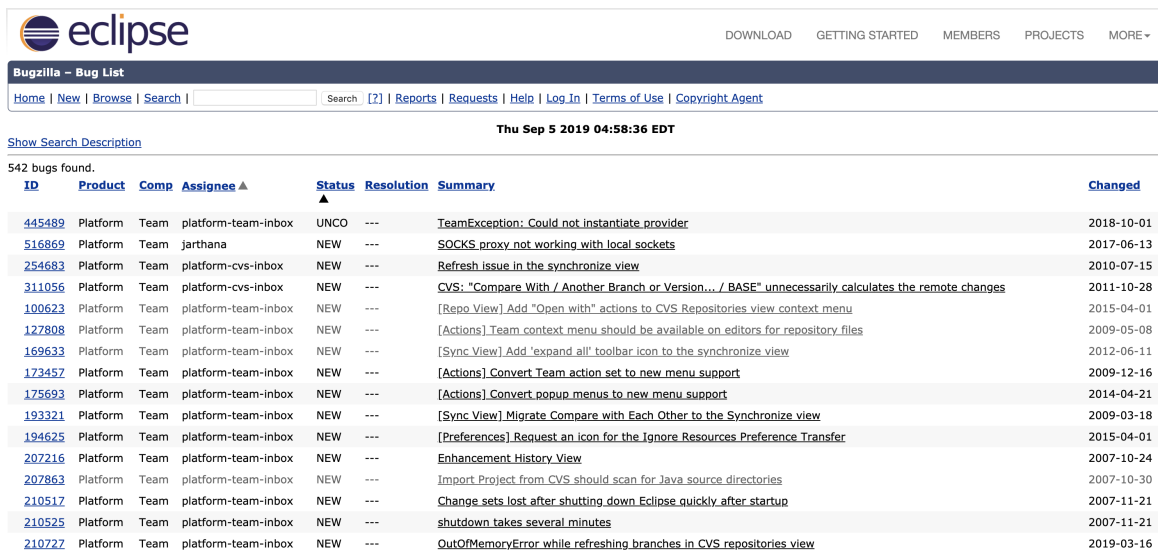
Figure 2.1: The example of bug reports in Bugzilla

2.1. BUG-TRIAGE USING BUG TRACKING SYSTEMS

release, managers inform the developers with a priority tag (The priority tag is shown at ⑤ in Figure 2.1).

After determining that the bug should be fixed, managers assign the bug to developers. The assigned developer develops a patch to fix the bug and then sends it to the project. Finally, the patch is verified by other developers and merged into the project's repository.

These activities (i.e., reading, prioritizing, and assigning bugs) are collectively called bug triage. Bug triage is a necessary activity in the process of fixing bugs. Several studies reported that bug-triage affects the time required for bug-fixing [32].



ID	Product	Comp	Assignee ▲	Status	Resolution	Summary	Changed
445489	Platform	Team	platform-team-inbox	UNCO	---	TeamException: Could not instantiate provider	2018-10-01
516869	Platform	Team	jarthana	NEW	---	SOCKS proxy not working with local sockets	2017-06-13
254683	Platform	Team	platform-cvs-inbox	NEW	---	Refresh issue in the synchronize view	2010-07-15
311056	Platform	Team	platform-cvs-inbox	NEW	---	CVS: "Compare With / Another Branch or Version..." / "BASE" unnecessarily calculates the remote changes	2011-10-28
100623	Platform	Team	platform-team-inbox	NEW	---	[Repo View] Add "Open with" actions to CVS Repositories view context menu	2015-04-01
127808	Platform	Team	platform-team-inbox	NEW	---	[Actions] Team context menu should be available on editors for repository files	2009-05-08
169633	Platform	Team	platform-team-inbox	NEW	---	[Sync View] Add 'expand all' toolbar icon to the synchronize view	2012-06-11
173457	Platform	Team	platform-team-inbox	NEW	---	[Actions] Convert Team action set to new menu support	2009-12-16
175693	Platform	Team	platform-team-inbox	NEW	---	[Actions] Convert popup menus to new menu support	2014-04-21
193321	Platform	Team	platform-team-inbox	NEW	---	[Sync View] Migrate Compare with Each Other to the Synchronize view	2009-03-18
194625	Platform	Team	platform-team-inbox	NEW	---	[Preferences] Request an icon for the Ignore Resources Preference Transfer	2015-04-01
207216	Platform	Team	platform-team-inbox	NEW	---	Enhancement History View	2007-10-24
207863	Platform	Team	platform-team-inbox	NEW	---	Import Project from CVS should scan for Java source directories	2007-10-30
210517	Platform	Team	platform-team-inbox	NEW	---	Change sets lost after shutting down Eclipse quickly after startup	2007-11-21
210525	Platform	Team	platform-team-inbox	NEW	---	shutdown takes several minutes	2007-11-21
210727	Platform	Team	platform-team-inbox	NEW	---	OutOfMemoryError while refreshing branches in CVS repositories view	2019-03-16

Figure 2.2: The list of reported bugs in Bugzilla

2.2 Bug Triage and Software Release

Software release is the process of delivering a new product that contains new functionalities or bug-fixes. In many cases, software development does not complete in a single iteration. Products are released in iterations to enhance them to satisfy customers or to adapt to the environmental changes.

Two strategies are known for releasing products, time-based strategy and feature-based strategy [33] [34]. Projects with feature-based release strategies first specify the functionalities to be implemented in the next release. Then, the developers implement the functionality and assure the quality by testing and reviewing. After all the features are completed, they can release the product to the public. Sometimes, deadlines are set up in advance but tend to be flexible. This strategy is often employed in product development in which all bugs are considered fatal and unacceptable, such as banking systems.

On the other hand, projects with time-based release strategies determine the release date in advance. The developers release only the functionalities that are ready by the release date. The developed products are released in short terms, such as 2–6 weeks, in order to adapt to changes in the market. Such release strategies tend to be accepted by projects that prefer delivering product value and receiving early feedback instead of troubling users with bugs (e.g., web-browsers). In open-source software development, Openstack and Firefox shifted to rapid release. Firefox had a time-based release strategy, every 3 months, but changed the development duration to 6 weeks so that it can compete with Google Chrome, which had already adopted rapid release.

Due to fierce competition in the market, modern software development tends to adopt time-based release, thereby forcing developers to complete not only feature implementations but also refactoring, testing, and bug-fixing before the tight deadline.

Such time-based release generates time pressure on developers, which would trigger prioritization, and bugs will be carried over to future releases. Figure 2.3 shows the overview of the bug triage process and the effect of release pressure. This is because projects cannot fix all the bugs by the next release date. Hence, developers need to prioritize bugs in order to decide which bugs should be fixed by the immediate release

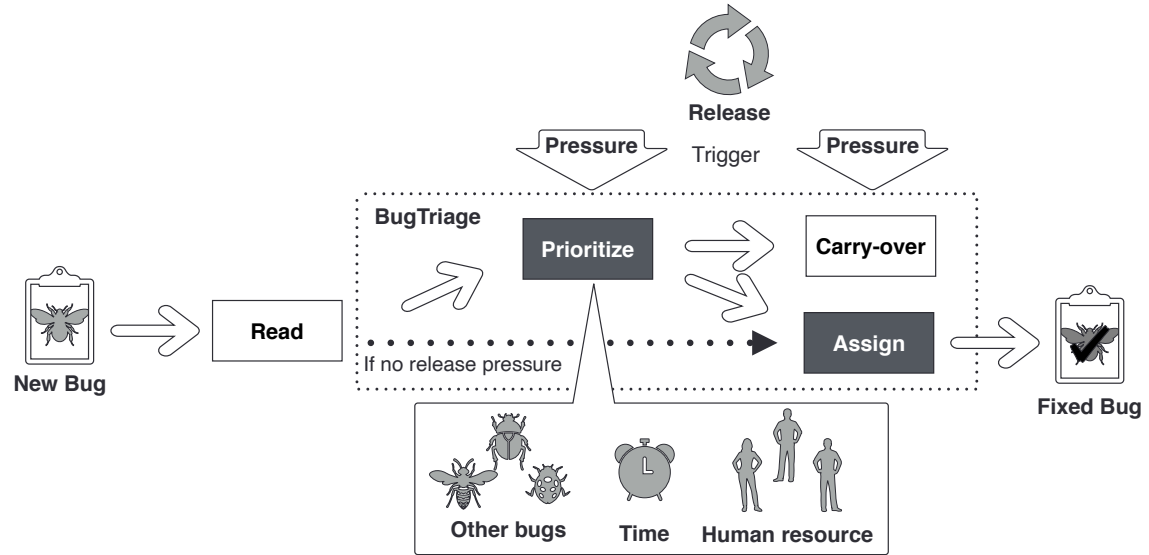


Figure 2.3: The process of the bug triage affected by release pressure

date and which ones should be carried over to future releases. To appropriately prioritize bugs, developers need to take into consideration not only the target bug but also the other unfixed bugs, the remaining time before the next release date, and the available human resources. If there are many unfixed bugs, less time before release, or fewer human resources, the number of bugs carried over to the future releases will increase. In modern software development, such prioritization and decision-making would play a crucial role in reducing the risk of annoying users with product bugs.

2.3 Bug Prioritization

2.3.1 Prioritization

Software projects receive a large number of bug reports every day [35]. Even if the existence of defects is confirmed, they will not always be fixed by the subsequent release. This is due to the fact that time and human resources available are limited. Developers must give utmost importance to fixing bugs that have greater impacts on their products and users. To share information on bugs that should be fixed fast, developers often utilize priority or severity tags prepared in bug reports. The levels of priority and severity are defined on the Eclipse project's web page ¹ (Shown in Table 2.1).

By definition, priority tags are set by developers to inform which defects should be fixed first whereas severity tags are set by reporters (including end-users and developers) to indicate how serious the impact of the defect is on their products, users, and development process. By looking into the descriptions of priority and severity levels, we can see that the priority levels are defined more ambiguously than the severity levels, which suggests us not to simply decide the priority. In order to decide the priority levels, developers might acquire various information such as not only the symptom of the bug to set priorities but also the situation of the project, etc. For example, developers need to read each bug report manually, and make comparison with other bugs whose priority is already set. Moreover, developers need to consider how many defects have not been fixed yet, how much human resources are available, and where the current progress is in their release cycle.

Thus, setting priorities is well known as time-consuming work [36]. Ideally speaking, even after the bugs are set a priority once, the priority should be regularly reviewed because various events happen in the project and the situation of the project varies day by day. However, this is impossible in reality because the reviewing priority will burden to the developers and developers have not already had enough time.

¹Eclipse Priority: https://wiki.eclipse.org/Eclipse/Bug_Tracking, Last Accessed: January 2020

Table 2.1: The levels of priority and severity defined on the Eclipse project web page

Priority	
P1	<i>“stop ship” defect i.e. we won’t ship if not fixed</i>
P2	<i>intent is to fix before shipping but we will not delay the milestone or release</i>
P3	<i>nice to have</i>
P4	<i>low priority</i>
P5	<i>lowest priority</i>

*“P1” is the highest and “P5” is the lowest priority.

Severity	
blocker	<i>the bug blocks development or testing of the build (for which there is no work-around)</i>
critical	<i>implies “loss of data” or frequent crashes or a severe memory leak</i>
major	<i>implies “major loss of function”</i>
normal	<i>default value, regular issue, some loss of functionality under specific circumstances, typically the correct setting unless one of the other levels fit</i>
minor	<i>something is wrong, but doesn’t affect function significantly or other problem where easy workaround is present</i>
trivial	<i>cosmetic problem like misspelled words or misaligned text, but doesn’t affect function (such as spelling errors in doc, etc.)</i>
enhancement	<i>request for enhancement (also for “major” features that would be really nice to have)</i>

*“blocker” is the highest and “trivial” is the lowest severity.

2.3.2 Related Work

To eliminate the effort of priority tagging, many studies have tackled automated priority predictions with machine learning algorithms [3, 36]. Kanwal et al. proposed a classification-based approach with Naive Bayes and Support Vector Machine (SVM) [36]. The proposed classifiers take a set of bug reports as an input which contains both categorical (e.g., severity) and text features, and known priority tags. In addition to the inherent features in bug reports, Tian et al. leverage various metrics from multiple factors such as project's daily activity factor (e.g., how many bugs are reported during a day), related-report factor (e.g., how high priority the similar bugs have), and so forth [3].

In this thesis, Chapter 4 addresses the same kind of priority predictions but focus on specific periods (i.e., testing and debugging phase). The previous studies use data collected from the whole development period without specification. However, priority is used for the next release. The usage of priority would vary depending on the type of release (e.g., major or minor) and the remained time before the next release date. Therefore, we build release-cycle aware models with the data in specific periods based on the release cycle.

However, several studies warn that priority/severity does not indicate accurate values because 65% of bug reports are mislabeled [26]. Hence, many studies have defined serious bugs that have a bigger impact on the product or process (which are named high impact bugs) [27, 28, 29, 30, 31] and build models to predict them. However, it is not sure whether developers consider these bugs as being highly impactful on the product or process. In the first step of this thesis, Chapter 3 conducts a survey to determine what kind of bugs developers think of being impactful. Then we try to find out whether those bugs are critical by conducting a practical investigation. Finally, we investigate whether the bugs that developers responded have higher importance.

2.4 Bug Assignments

2.4.1 Manual Bug Assignments and Its Limitation

While a myriad of bugs are daily reported in projects, managers dedicatedly read each of them to assign it to a developer so they could be assigned to their matching developer. It is hard to select an appropriate developer because there are not only numerous bugs but also developers (especially, during the coding and testing phases). Particularly, the managers face difficulties while trying to decide on aspects such as the set of skills each developer has and the number of bugs each developer is currently addressing.

These difficulties would often lead to reassignments. This happens when the bug is assigned to another developer from the first assigned one because he/she cannot fix the bug. One reassignment induces an additional delay of 50 days during the bug-fixing process in Eclipse Platform and Mozilla Firefox. The reassigned bugs account for about 40% of all bug fixes [11]. Reassignments should be prevented as much as possible because not only do they waste human resources but also delay the bug-fixing [13].

2.4.2 Related Work

In order to assign bugs to their appropriate developers or reduce the amount of effort coming from that procedure, many methods have been proposed over a decade [2, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22]. We summarize bug assignment methods in Table 2.2 and below.

Expertise-aware methods aim to assign bugs to the developer who has appropriate expertise which is calculated from similar bugs that developers previously fixed [2]. The similarity of bug reports is measured from the description present in the bug reports [2, 21, 35, 37, 38, 39, 40, 41, 42] or source code history [14, 20, 43, 44]. Anvik and Murphy built a classifier (e.g., Naive Bayes [45], SVM [46]) with the sets of words in the bug report and the name of the developer who fixed the bug [35]. The model can recommend developers who are capable of dealing with newly reported bugs with relatively high accuracy (about 70%-75%).

Activeness-aware methods try to assign bugs to active developers [21, 39, 44, 47]. Wang et al. measured developers’ activity scores in each component for a few months and built a method assigning bugs to the active developer who has the highest score in the component that the bug involves [47]. This method does not need training classifiers but has improved about 20% of the assignment accuracy compared to expertise-aware techniques.

Experience-aware methods aim to assign bugs to developers who have contributed to projects [17, 22, 48]. Naguib et al. have proposed a method to rank developers based on the times of bug fixing activities (e.g., number of fixed bugs, number of comments, and so forth) [17]. The method achieved an average accuracy of 88% with the top 10 recommendations and outperformed the expertise-aware method [40].

Cost-aware methods aim to reduce bug fixing time, to keep the accuracy of assignments [18, 22, 23]. Park et al. have extended Anvik’s method [35] and presented CosTriage which takes the cost of bug-fixing into consideration [18]. CosTriage requires estimating the cost of the time to fix each bug. It is calculated by using the average time to fix similar bugs while taking into account the possibility of the assignment being appropriate (calculated in the same way as Anvik’s method). Costriage assigns a bug to the most appropriate developer. This method is based on the ratio of the accuracy level compared to the bug-fixing speed (determined beforehand). While the accuracy decreases by approximately 5% compared with Anvik’s method, Costriage can reduce 7%-31% of the average bug-fixing time.

Importance-aware methods consider the levels of priority or severity contained in bug reports which shows the importance of fixing the bug [15, 48, 49]. Priority and severity levels show how important it is to fix bugs for developers and users, respectively. Lin et al. have built a model considering priority and severity in addition to text data, which was used to conduct an empirical study with the data including Chinese characters and showed non-textual data is comparable to textual data [15]. These approaches are close to ours, but still different. They are in fact made for assigning the appropriate developers. Our methods are mainly focused on decision-making (i.e., which bugs should be fixed by the next release date).

Release-aware methods aim to assign bugs so that the amount fixed by the next release date can be increased. Although the other methods (described above) have the advantages of recommending appropriate developers or reducing bug-fixing time, those methods tend to assign bugs to the few most active developers [18, 23] (we will call this the “task concentration problem”). Even experienced developers can only fix a limited number of bugs. In fact, most projects have short release cycles with usually limited remaining time before the next release. This is detrimental, and thus should be taken into account when developing methods. To the best of our knowledge, there are no assignment methods that consider releases. In this thesis, Chapter 5 tries to build a release-aware method in order to increase the number of bugs before the next release. We place a limitation on the number of tasks that are assigned to each developer during a given period; the method assigns bugs under the constraint while considering developers’ skills.

Table 2.2: An overview of prior studies about bug-assignment methods

Paper	Types of the awareness for bug-assignment methods					Dataset
	Expertise	Activeness	Experience	Cost	Importance	
Cubrani; 2004 [37]	✓					Eclipse
Anvik; 2006 [35]	✓					Eclipse, Firefox, Gcc
Lin; 2009 [15]	✓				✓	SoftPM
Anvik; 2011 [2]	✓					Eclipse, Firefox, Gcc, Mylyn, Bugzilla
Wu; 2011 [41]	✓					Firefox
Tamrawi; 2011 [21]	✓	✓				FireFox, Eclipse, Apache, Netbeans, FreeDesktop, Gcc, Jazz
Park; 2011 [18]	✓			✓		Apache, Eclipse, Linux, Mozilla
Linares-Vasquez; 2012 [43]	✓					ArgoUML, JEdit, MuCommander
Servant; 2012 [44]	✓	✓				iBugs
Somasundaram; 2012 [40]	✓		✓			Eclipse, Mylyn, Mozilla
Kumar Nagwani; 2012 [48]			✓		✓	Mozilla
Naguib; 2013 [17]	✓		✓			Atlas, Birt, Unica
Zhang; 2013 [22]	✓		✓	✓		Eclipse, JBoss
Wang; 2014 [47]		✓	✓			Eclipse
Xia; 2015 [42]	✓					GCC, OpenOffice, Mozilla, Netbeans, Eclipse
Shokripour; 2015 [39]	✓	✓				Eclipse, Netbeans, ArgoUML
Park; 2016 [23]	✓			✓		Apache, Eclipse, Linux, Mozilla
Lee; 2017 [38]	✓					Eclipse, Firefox, Industrial
Sharma; 2017 [49]	✓				✓	Firefox, Bugzilla

Chapter 3

A Survey Study of the Bug-fixing Importance

3.1 Introduction

Several studies have proposed methods to predict the priority of bugs reports [3, 36]. However, Saha et al. warn that priority/severity does not reflect the actual values that should show because 65% of bug reports are mislabeled [26]. Hence, many studies have defined serious bugs that impact on the product or process (which are named high impact bugs) [27, 28, 29, 30, 31] and build models to predict them. Still, it is still unsure which/what bugs developers think are highly impactful on the product or process for developers.

In this study, we ask what kind of bugs developers think are impactful and encounter in practice, through a survey with 322 notable GitHub developers. To the best of our knowledge, this work is the first survey focusing on only high impact bugs. This chapter addresses three research questions as follows;

RQ1: What kinds of high impact bugs are mainly considered high impact by OSS developers?

With a closed question, we asked developers which bugs that prior work has studied is highly impactful. We found that security bugs are the most frequent answer.

RQ2: What kinds of high impact bugs do OSS developers encounter most frequently?

With an open question, we asked developers what high impact bugs developers encounter so far. We manually classified the responses including free text and found that breakage bugs are the most frequent and the security bugs is following.

RQ3: Do high impact bugs have a higher importance?

With the bugs that developers responded as high impact, we investigate if they have higher importance. We found that 66% of the bugs have higher priority or severity than the default level.

Chapter Organization: The rest of this chapter is organized as follows. Section 2 introduces some existing studies on finding and fixing high impact bugs. Section 3 describes the study design and Section 4 shows the survey and classification results. We discuss the results in Section 5. Finally, Section 6 concludes this chapter.

3.2 High Impact Bug

Over the past two decades, the SE community have dedicated considerable efforts to help software developers to predict faults in modules, localize and repair faults in source code, and so on. Although the traditional studies had not thoroughly considered the characteristics nor the impacts of bugs, in recent years they began to tackle the impacts of bugs on users and the development process. In what follows, we introduce some existing studies on finding and fixing high impact bugs.

A SECURITY BUG [27] can cause serious problems that often impact on uses of software products directly. Since Internet devices (e.g., smartphones) usage is increasing every year, security issues of software products are of interest to many people. In general, security bugs are fixed as soon as possible.

A PERFORMANCE BUG [28] is defined as “programming errors that cause significant performance degradation.” The performance degradation contains poor user experience, laggy application responsiveness, lower system throughput, and waste computational resources [50]. [28] showed that a performance bug needs more time to be fixed compared with a non-performance bug.

A BREAKAGE BUG [29] is a type of functional bug which is introduced into a product when the source code is modified to add new features or to fix existing bugs. Though it is well-known as regression, a breakage bug mainly focuses on regression in functionalities. A breakage bug causes problems that make available functions in one version unusable after releasing newer versions.

A BLOCKING BUG [30] is a bug that prevents other bugs from being fixed. It is often caused due to a dependency relationship among software components. Since a blocking bug inhibit developers from fixing other dependent bugs, it can highly impact on developers’ task scheduling since a blocking bug takes more time to be fixed [30] (i.e., a developer would need more time to fix a blocking bug and other developers need to wait for being fixed to resolve the dependent bugs).

A SURPRISE BUG [29] is a new concept of software bugs. It can disturb the workflow and/or task scheduling of developers since it appears at unexpected times (e.g., bugs detected in post-release) and locations (e.g., bugs found in files are rarely changed in pre-release). As a result of a case study that uses a dataset of a proprietary, telephony system which has been developed for 30 years, [29] showed that the co-changed files and the amount of time between the latest pre-release date and the release date can be good indicators of predicting surprise bugs.

A DORMANT BUG [31] is also a new concept on software bugs and is defined as “a bug that was introduced in one version (e.g., Version 1.1) of a system, yet it is not reported until after the next immediate version (i.e., a bug is reported against Version 1.2 or later).” [31] showed that 33% of the reported bugs in Apache Software Foundation projects were dormant bugs and were fixed faster than non-dormant bugs.

3.3 Study Design

3.3.1 Overview

In this study, we e-mail and ask notable developers in GitHub¹ to answer a questionnaire about high impact bugs. After aggregating collected responses, we show the developers' demographic information (**Q1**), and the distribution of the bugs that are considered high impact (**Q2-1**). As the questionnaire includes an open question (**Q2-2**) to tell us actual bug reports which caused troubles in the past, we manually inspect the bug reports and categorize them by symptoms. Finally, we access the links of high impact bugs and investigate if their importances in the bug reports are higher or not.

3.3.2 Participant Selection

In order to select notable developers to invite to our survey in this study, we use *contribution* which represents the amount of the developer's commit activity to GitHub repositories and can be calculated with GitHub API ². First, we make a list of all repositories in GitHub and calculate the total number of contributions for each repository. Note that we only calculate contributions for the most committed repositories if the repositories have the same name, since forked repositories partly (sometimes largely) include the same commits from original repositories and we need to avoid multiple counts for the same contributions by the same developer. Next, the total contributions of each developer is calculated based on the selected repositories above, and we choose candidates who mark over 100 contributions. Finally, we sent e-mails to 22,228 candidate developers to ask them to participate in our survey.

¹Github: <https://github.com/>, Last Accessed: January 2020

²GitHub API: <https://developer.github.com/v3/>, Last Accessed: January 2020

Table 3.1: The questionnaire for our survey

[Q1. Profile]	
Q1-1	Your main project
Q1-2	Your experience with OSS development
Q1-3	Your motivation to participate in OSS development
[Q2. High impact bugs]	
Q2-1	What kind of bug would be much more important to be fixed? <ul style="list-style-type: none"> - a bug threatening systems' security (Security bug) - a bug deteriorating system's performance (Performance bug) - a bug blocking other bug fixes (Blocking Bug) - a bug found in unexpected timing and location (Surprise bug) - a bug introduced in older releases and found in a newer releases (Dormant bug) - a bug introduced in a newer release and breaking functions of older releases (Breakage bug) - others [free text]
Q2-2	Please tell us high impact bug(s) you encountered in the past.

3.3.3 Questionnaire

We prepare Google Forms for our survey which consisted of three questions to know the developers demography (**Q1**), one closed question to reveal a distribution of high impact bugs considered important by developers (**Q2-1**), and one open question to collect and further analyze actual reports on high impact bugs (**Q2-2**). The detail contents are shown in Table 3.1.

3.3.4 Categorization of Bug Symptoms

Based on the responses of **Q2-2**, we collect actual bug reports from developers' projects and confirm the symptoms of the bugs, in order to discuss what techniques have been already proposed or that will be required to find and fix those high impact bugs. The first and second authors independently and manually inspect symptoms of actual high impact bugs and classify them by the card sort technique [51]. After the independent classification, the two authors discuss each classification result together and merge the results by mutual consent to make one classification. Here, the inspectors include one

Ph.D. student (first author), who worked at a software company for two years as a full-time developer, and one professor who has been studying OSS development over ten years.

3.4 Survey and Classification Results

3.4.1 Developer Demography (Q1)

As we described earlier, we invited 22,228 developers to join our survey. During the two weeks survey period, we got responses from 322 developers. Table 3.2 shows the product domains where the developers participated. We can see “web application” is the most popular domain (7%) but it does not stand out from the others. The product domains spread across a wide area. We can assume that the results of our survey reflect a wide range of situations across OSS development. Table 3.3 shows the developers’ experience with OSS development. The majority of the developers have over five years experiences. It is no surprise because we only invite active developers who have made at least over 100 commits to GitHub repositories. Table 3.4 shows developers’ motivations to OSS development. 59% (126+64) of the developers contribute to OSS projects as part of work.

3.4.2 RQ1: What Kinds Of High Impact Bugs Are Mainly Considered High Impact By OSS Developers?

In Q2-1, we asked the developers to select one of the six kinds of high impact bugs which are introduced in the previous section and have been well-studied in the SE community. Table 3.5 shows the responses from the developers. We can see the OSS developers from GitHub attach greater importance on security bugs (53%) and breakage bugs (22%). It was unexpected for us that the other four bugs attract less attention from the OSS developers. It partly indicates the perception of gaps between researchers and OSS developers. Some researchers in the SE community might misunderstand OSS developers’ actual needs.

3.4. SURVEY AND CLASSIFICATION RESULTS

Table 3.2: Product domains where GitHub developers join (Q1-1)

Domain	#	%		Domain	#	%		Domain	#	%
web application	22	7%		database	9	3%		machine learning	5	2%
development tool	19	6%		network server	9	3%		UI	5	2%
analysis	19	6%		messaging tool	9	3%		mobile app	5	2%
language & compiler	17	5%		education	8	2%		desktop system	4	1%
OS	15	5%		simulator	7	2%		mail	4	1%
graphic	14	4%		finance	7	2%		browser	3	1%
game	13	4%		resource monitoring	7	2%		others	37	11%
programming tool	12	4%		image editor	6	2%		no answer	34	11%
blog	11	3%		network tool	6	2%				
embedded OS	9	3%		security tool	6	2%		Total	322	100

Table 3.3: Experience with OSS development (Q1-2)

Experience	Developers
more than five years	213
three to five years	63
one to three years	45
less than one year	1

Table 3.4: Motivation to participate in OSS development (Q1-3)

Motivation	Developers
hobby	111
work	126
both	64
other	21

Table 3.5: A distribution of high impact bugs (Q2-1)

high impact bugs	#	%
Security bug	171	53%
Breakage bug	72	22%
Performance bug	20	6%
Blocking bug	16	5%
Dormant bug	12	4%
Surprise bug	7	2%
others	24	7%

Researchers in the SE community have been studying to help developers find and fix bugs especially in terms of impacts on users' satisfaction and during the development process (release) [52]. Although high impact bugs have been studied individually so far, it was not clear if OSS developers are mostly concerned with particular high impact bugs. From the result of Q2-1, we now answer RQ1 as follows.

Answer to RQ1: Researchers have been dedicating to provide a means to find and fix a variety of high impact bugs, but OSS developers mainly emphasize a focus on security and breakage bugs.

3.4.3 RQ2: What Kinds Of High Impact Bugs Do OSS Developers Encounter Most Frequently?

In Q2-2, we asked the developers to describe the high impact bugs they have encountered in the past. Many of the developers described characteristics of high impact bugs in the free text format and also gave us direct links to actual bug reports which present symptoms discussed among developers and users.

Table 3.6 shows symptoms of bugs considered high impact by the respondents (Ref in the table will be used in the discussion section). In the table, we count multiple times if a developer described several high impact bugs. The percentage in the table is the ratio of developers' answers in each category, but the total percentage does not become 100% due to the above reason. As we described earlier, we manually inspected and categorized the information on high impact bugs by symptom. In what follows, we summarize the classification result.

We had 249 valid answers from 192 developers about symptoms of high impact bugs which actually get OSS developers in trouble in the past. In Table 3.6, the most common symptom was "unexpected processing" responded by 17% of developers (42 cases). As regards "unexpected processing", we could confirm less in common with bug reports. They ranged from different calculation results to unexpected rendering. The next most frequent was "sudden stop" responded by 16% of developers (39 cases). The corresponding bug reports shown by the developers suggested us that it often happened due to null pointer

3.4. SURVEY AND CLASSIFICATION RESULTS

Table 3.6: The number of categorization based on the symptoms in actual bug reports. The related researches are shown in Ref column.

Category	Subcategory	Description	#	%	Ref
Behavior	disable start	Users cannot install, compile or start an application	19	8%	[53][54][55]
	never start function	A function never start once a user clicks a button	21	8%	[29][56][57][58]
	sudden stop	A program suddenly stops during running	39	16%	[24][59][60][61]
	unexpected processing	A program does not output or behave as developers expected	42	17%	[57][62]
	never finishing state	A process never finish (e.g, hang up and infinite loop)	5	2%	
	unable to login	Users cannot login a system	4	2%	
	others	Lack of items in display, wrong warnings, lower user experiences etc.	8	3%	
Effect	lower performance	A program lowers performance (e.g, too large memory usage)	13	5%	[28][63][64][65]
	damage other systems	A program damages other systems (e.g, OS cannot boot)	5	2%	
	others	Making a disk full etc.	3	1%	
Security	vulnerability	Security defects allow an attack to cause an abnormal behavior	22	9%	[66][67][68]
	unauthorized access	An impersonating account accesses to a server, service, or data	28	11%	[69][70][71]
	DDoS	Massive accesses from multiple terminals make a service unable	7	3%	
Data	data loss	A program deletes data in a product (e.g., user data and database breakage)	12	5%	
	incorrect data	A program produces incorrect or duplicated data	1	0%	
Development	architecture change	It forces developers to change a architecture or core program in a product	3	1%	
	reproduce	Developers cannot reproduce a reported bug	3	1%	
	others	Blocking other bugs fixed etc.	4	2%	
Reputation	compatibility	Compatibility is broken (e.g, API, hardware and OS)	7	3%	
	execution env.	A product can not guarantee an execution environment	3	1%	

3.4. SURVEY AND CLASSIFICATION RESULTS

exception, run-time error exception, segmentation error, and overflow. Although the above two are related to “Behavior” of a program, the third and fourth most common symptoms were “Security” concerns such as “vulnerability” and “unauthorized access.” About “vulnerability,” the corresponding bug reports suggested the developers especially concerned with XSS and SQL injection attacks. The OpenSSL problem (i.e., Heartbleed) and the hidden way of leaking user data were included in bug reports about “unauthorized access.”

In RQ1, 53% of developers think that security is the biggest concern among high impact bugs in the previous studies. However, the result in RQ2 shows that the developers come across high impact bugs about behaviors more often than the one about security. In fact, one developer said, *“Since the mentioned project is (mostly) a client-side javascript library, security problems aren’t common.”* Based on the results here, we answer RQ2 as follows.

<p>Answer to RQ2: OSS developers most frequently encounter bugs relating to behaviors such as unexpected behaviors and sudden stops. Security bugs such as vulnerabilities and unauthorized accesses are also often encountered.</p>

3.4.4 RQ3: Do High Impact Bugs Have a Higher Importance?

RQ2 showed that there are many kinds of high impact bugs. This variety requires substantial effort to build models to detect each kind of bugs. Hence, we need to find easier ways to distinguish bugs. Several studies utilize some attributes to represent importance (e.g., severity, priority) [3, 36, 72] while others argue that these are unreliable [26]. Therefore, we need to investigate if the severity and priority tags are useful for detecting high impact bugs. In RQ3, we access the links of high impact bugs from the responses and investigate what percent of them have a higher importance. Investigating the importance is close to the previous study [26] but we examine the high impact bugs admitted by the respondents whereas they randomly selected the bugs.

As a result, we could access 136 bugs and found 68 bugs have any priority or severity placements. We investigated the importance of 68 bugs and found that 45 bugs (66%) have a higher importance, which is not a small number of bugs because the percentage is usually low [3, 36]. This suggests priority and severity might help to detect high impact bugs.

Answer to RQ3: 66% of the bugs have a higher importance than the default level where projects provide an importance label in BTS.
--

3.5 Discussions

3.5.1 What Kinds of High Impact Bugs Should Be Studied Newly by the SE Community in Order to Support OSS Developers?

In this study, we found there are a variety of high impact bugs. However, we still do not know whether these bugs are studied or not. In this section, we investigate if prior work covers the high impact bugs that developers' responded.

The percentages in Table 3.6 are indicated in boldface if the corresponding symptoms account for about 80% of all the symptoms (i.e., the developers frequently encounter the symptoms with boldface.). For the majority of the symptoms, we surveyed existing studies which have tried to find or resolve the symptoms and showed references as "Ref" in Table 3.6.

The percentages of the symptoms in "Behavior" category are relatively high and these have been well-studied by the SE community [24, 29, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62]. For instance, "never start function" is well studied *breakage bugs* [29] so-called regressions which disable existing functions due to additional changes to software products. Although in the chapter we did not introduce this as a high impact bug, "unexpected processing" is well studied as a functionality bug or feature bug [57]. "disable start" and "sudden stop" are also studied as build process bugs [55] and crash bugs [24], respectively.

As we confirmed "vulnerability" and "unauthorized access" achieved relatively high attention from the developers, *security bugs* are also considered high impact by researchers and have been well studied [65, 66, 67, 68, 69, 70, 71]. "Lower performance" in "Effect" category is also well studied [28, 63, 64, 65] as performance bugs. However, to the best of our knowledge, there is no study on "data loss" in "Data" category which is of relatively high concern to OSS developers (5%). For instance, a bug on "data loss" in "Data" category is observed when deleting data related to the operation under a condition. Other data loss bugs occurred due to executing the wrong processing of multi-transaction or by using variables not multi-threaded (e.g., HashMap in Java). In fact, loss of users' data

such as their pictures was recently reported in the update of Windows 10³. We regard it is one of the perception gaps between OSS developers and SE researchers and should be studied, allowing us to address the issue.

3.5.2 Threats to Validity

Internal validity: The categorization of Table 3.6 may not be perfect. We have a good deal of knowledge about software, but we also recognize the limitations of our knowledge about specific domains. We also might bias in creating the category.

External validity: Although the developer demography consists of developers working in a wide range of product domains, a judgment if a bug has high impact or not would depend on a product domain.

Construct validity: To avoid bias in the developers responses, we asked them about high impact bugs without providing rigorous definitions of high impact bugs. Attitudes towards high impact bugs might be different among the developers.

³Ars Technica: <https://arstechnica.com/gadgets/2018/10/microsoft-suspends-distribution-of-latest-windows-10-update-over-data-loss-bug/>, Last Accessed: January 2020

3.6 Chapter Summary

Several studies have proposed methods to predict the priority of bugs reports. However, Saha et al. warn that priority/severity does not reflect the actual values that should show because 65% of bug reports are mislabeled [26]. Hence, many studies have defined serious bugs that impact on the product or process (which are named high impact bugs) [27, 28, 29, 30, 31] and build models to predict them. However, it is not sure whether developers consider these bugs as being highly impactful on the product or process.

In this study, we asked what kind of bugs developers think are impactful and encounter in practice, through a survey with 322 notable GitHub developers. We inspected the responses and manually classified the bugs that affect seriously the users or process, included in the responses. As a result, we found that security and breakage bugs are highly important for OSS developers. Furthermore, we investigated if the high impact bugs, which developers responded, have higher importance and found 66% of the bugs have higher priority or severity than the default level, indicating that the importance of bugs help us to detect high impact bugs.

Chapter 4

Release-aware Bug Priority Prediction

4.1 Introduction

Many studies dedicate considerable effort to supporting bug management or bug fixing process, proposing various prediction methods such as bug prediction [73], bug-fixing time prediction [74], bug priority/severity prediction [3, 36, 72, 75]. Traditionally, to build priority/severity prediction models, most studies utilize all the accessible data or randomly chosen data [3, 36, 72, 75]. For example, we investigated the basis of datasets in priority and severity prediction studies during their generation and found 13 priority prediction studies and 24 severity prediction studies in total. Out of them, 6 and 14 studies (46% and 58% of the studies) used data during defined periods (e.g., 1/Jan/2001 – 31/Dec/2006 from Eclipse) in priority and severity predictions, respectively. Also, these studies advance no reason for choosing the defined period. Further, 2 and 8 studies (15% and 33% of the studies) lack details of datasets (e.g., these only mention the use of Firefox or commercial software) in the priority and severity studies, respectively. 5 priority studies (38% of the studies) randomly selected data from bug tracking systems (none of the severity studies). 2 severity studies (8% of the studies) collected data based on a specific software version (e.g., Eclipse 3.1) while none of the priority studies created

datasets based on versions.

Most of the studies indicate unawareness of the presence of releases, implying these unwittingly assume uniform activities **within** iterations ¹ (i.e., performing the same task in every day) and **across** iterations (conducting an identical task in each iteration), which is frequently untrue. Some studies [76, 77, 78] raise the concern of changing properties of data with time (across iterations), which causes lower performance of prediction models over time [79, 80]. For example, modern software development projects (e.g., agile software development) add new features in products for short periods. Even if during the short period, projects follow the software development process (i.e., planning, designing, implementing, and testing), the activity at any point in this period differs.

Recently, the assumption in software development projects are more difficult to satisfy. To facilitate multiple agile teams working in parallel, many large companies or OSS projects adopt the release train practice which coordinates releases across multiple teams and delivers the products at the same time with fixed and strict schedules. Consequently, shortening the release cycle is now frequent, causing developers to switch development phases in shorter than usual periods. This creates datasets to include multiple iterations of the planning, designing, implementing, and testing phases. Prediction models from such datasets are opaque because these are unaligned with the sequence of events, thereby reducing the performance. To the best of our knowledge, there is no empirical evidence and the impact remains unclear, although Adams and McIntosh suggest the anticipated impacts on software engineering studies by the release cycle [81].

This chapter examines the impact of the release cycle alignment on defect management prediction, especially, bug priority prediction. This is the focus because priority prediction is a very important field, whereas studies till date focused on defect prediction. We propose release cycle-aware models trained on data in a defined period, Although the original dataset contains multiple releases, we split it into several subsets aligned with the major releases and measure the impact of the alignment. Moreover, we compare the prediction performance while changing the datasets granularity to produce finer-grained

¹Iterations are fixed-length timeboxes in software development, and projects usually contain the aims in the iteration at the beginning of iterations.

analysis/models.

To investigate the impact of the release cycle alignment on the defect management process, we also address the following research questions (RQs) in this chapter:

RQ1. Do defect management activities vary within the release cycles?

We found that various bug fixing activities and the frequency of priority setting vary during the release cycle. We also observed two types of metrics vary or are invariable within the release cycle, with 42% of the metrics showing significant differences during specific periods.

RQ2. Do cycle-aware models outperform the cycle-unaware models?

Based on the month of the bug reported date while disregarding the year, we aggregated the data quarterly datasets from July (first month after main release). For each dataset, we created release cycle-aware models (CYC) and compared them with cycle-unaware models built using full data. The results show that the CYC outperforms the cycle-unaware model for recall and the g-mean.

RQ3. What is the right granularity for cycle-aware models?

In addition to the 1-divided models and 4-divided models (Quarter models) in the RQ2, we built 12-divided models (named monthly models) and compared them. The finer-grained models exhibited better performance than the coarse-grained models.

The contributions of this study are the following: (1) demonstrated that developers' activities vary during the release cycle; (2) showed that 42% of metrics are statistically significant different for specific periods due to variations in the release cycle; (3) clarified that the cycle-aware model exploits the characteristics of specific periods, and therefore, outperform the cycle-unaware model; and (4) proved that finer-grained data yield more accurate prediction models.

Chapter Organization: The remainder of this chapter is organized as follows: Section 2 provides a background on priority setting and release engineering; Section 3 contains a description of the design of the case study; Section 4 involves presentation of results from our four research questions; Section 5 contains a discussion of the application of various insights for predictions and disclosure of threats to the validity of our study; the chapter terminates with conclusions in Section 6.

4.2 Release Engineering and Bug Management

4.2.1 Release Engineering

Although all software projects involve a release cycle, the time or nature of the release (major, minor, or patch) differs among projects, since projects are based on different policies (e.g., extent of quality assurance) in various environments (e.g., proportion of market share retained). Since the release cycle differences affect the understanding of clues or lower the performance of prediction models in software engineering studies, researchers must be aware that the release cycle time or nature of the release affects developers' activities or practice [81]. We describe how the release cycle time or the nature of release affects the developers' activity or practice as bellow.

Release cycle time: Most software projects involve time-based release schedules (e.g., weekly or monthly releases) or feature-based schedules (e.g., immediately every feature is implemented) for release of the product. For instance, popular software projects such as Eclipse, Firefox, and Node.js adopt time-based release schedules. These involve the release a new product approximately every 3-5 months for Eclipse, 6-8 weeks for Firefox, and 6 months for Node.js.

Time-based projects optimize the periods before the next release (release cycle time) for adapting to factors like the competition environment and their policy. For example, Firefox conducts seven major releases, while Eclipse is characterized by one main release and two service releases annually. Firefox is required to rapidly deliver new features, upgraded security, stability, and other bug fixes to compete with Google Chrome [82]. Therefore, Firefox adopted a rapid release cycle from version 5.0, delivering a new product every 6-8 weeks [83]. Conversely, Eclipse depends on a longer release cycle to ensure backward compatibility, stability, and smooth transitions, spending more time for release planning [84].

The difference in the release cycle time can affect software development. Consider Project A with a release cycle of 2 months and Project B with a cycle of 4 months, respectively involve 5 reports daily. Project A requires dealing with about 300 reports before the next release, whereas Project B must handle 600 reports. Therefore, the project

with the longer release cycle caters for more defects. The longer release cycle commonly involves changes reflective of the additional source code that emerges before the next release. Therefore, Adams and McIntosh indicate that researchers should avoid simple comparison of the number of reports, commits, and other overlapping characteristics between projects with different release cycles [81].

Nature of releases: Most projects adopt the semantic versioning [85], defining a guaranteed level of compatibility. In general, the semantic versioning represents the level as three digits (e.g., X, Y, and Z). The first digit (X) represents a version of the major release with incompatible API changes. The second digit (Y) and the last digit (Z) denote a version of a minor release and patch release (bug fixes) changes with backward compatibility, respectively.

In Eclipse, end users are provided a main release (either major release [e.g., Eclipse 3.0] or minor release [e.g., Eclipse 3.1]) in June each year. Also, end users are furnished with two service releases (SR1 and SR2) in September and February each year, with new features included in the main release. The patches for fixing bugs in the main release are provided in the two service releases. Firefox releases major versions every 6-8 weeks for end users in four phases. Before delivering a main release, three channels are prepared (i.e., *Nightly*, *Developer Edition*, and *Beta*), so that the product can be tested by developers or corporate users [86]. New features are developed from the *Nightly* edition, tested on the Developer edition and Beta before provision to end users in the main release as a final product.

Therefore, every project involves a unique nature of release (major, minor, or patch release) for improving the quality of the product. Depending on the nature of the release, the developers' activity would obviously vary. Before a major release, focus may be on developing new features involving adding new code to repository. A minor or patch release usually involves small changes in the existing code while retaining backward compatibility. These differences in activities influence various predictions, especially priority predictions, because priority labels are usually set for the next release. For example, if the release (Eclipse: main release, Firefox: Nightly release) includes many new features, enhancement reports may be prioritized. Contrarily, the bug-fixing task may be prioritized when a

release (Eclipse: service releases, Firefox: alpha release) includes many patches for defects. Although releases included in datasets require careful consideration, no study related to this exist or consistent attention on this is lacking.

4.2.2 Bug Management and Release

In the mining software repository (MSR) field, many studies that devoted time on bug prediction, bug fixing-time prediction, or bug priority prediction for improving the bug fixing process exists. Most of studies assume that data in a period is always similar those in other periods, using all available data to generate prediction models. Against this assumption, several studies raise concern that data trends in a project change over time according to the system’s age, growth, etc. [76, 77, 78]. McIntosh and Kamei built fine-grained prediction models for identifying fix-inducing code changes, showing that data immediately before the prediction time, contributes the most to the prediction performance [77]. Moreover, they also revealed accuracy fluctuations and variations in important metrics of the prediction models across test periods.

These fluctuations are probably affected by the release cycle. Since developers focus on new features during specific months and on bug fixing during other months, important metrics required for prediction usually changed. Using all data in release cycles combines the characteristics of each period among the release cycles, missing the subtle, and thereby diminishing the accuracy of predictions. Thus, we believe that SE researchers must carefully consider trend changes over time and the trend during the release cycle.

In this study, we investigate the impact of release cycles on the bug fixing process, specifically for bug priority prediction. We also build cycle-aware models with data during some or divided periods, evaluate these using the Eclipse Platform dataset to compare the performance of CYC versus cycle-unaware models.

In Figure 4.1, we illustrate the concept of the release cycle-aware models with examples. The left side of Figure 4.1 shows that the project involves three teams with different release cycles and is characterized by time-based releases. Team A and Team C iterate two and three iterations per release, respectively but the second iteration of Team C is longer than the others (similar to Eclipse 3.x projects). Conversely, Team C releases a

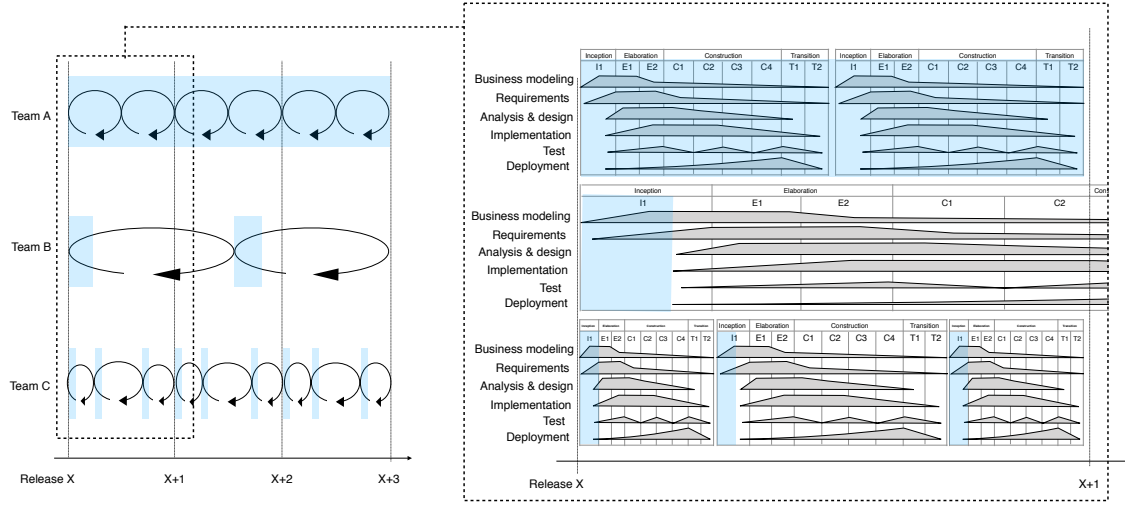


Figure 4.1: Concept of the cycle-aware models. The cycle-aware models are trained with data in specific periods of the release cycle (e.g., the periods after releases).

product once in three release dates and involves two iterations by the time release time of a product.

The task conducted according to the period in the iterations are shown in the right side of Figure 4.1, with each graph based on the Rational Unified process [87]. The blue shades represent the extent of data used for building the prediction model, with three extents prepared. Particularly, the graph for Team A in the right side of Figure 4.1 involves the use of all data for building the models. Evidently, the blue shade covers all tasks, indicating that models are built irrespective of the tasks performed by developers, reflective of the method involved in traditional studies.

The blue shades for Teams B and C in the left of Figure 4.1 reveal that a focus to the left of the cycle. Moreover, looking at the right side of Figure 4.1, the covered tasks essentially involve business modeling and requirements. Based on data highlighted by the blue shade, the prediction models can be rendered specialized for business modeling and requirements. Considering priority predictions currently, the prediction model learns bugs that should be prioritized for business modeling and requirements, granted that some

bugs reporting about others (e.g., design or architecture) would not be prioritized.

Thus, data must be acquired from an appropriate period for prediction of the target task, and notably, the granularity of the dataset requires consideration. Suppose we apply the release cycle-aware model of Team B to Team C, this will no longer work because it contains data for various tasks as well as Team A. Excess splitting creates lack of data for the prediction model, thereby reducing the prediction performance. In this study, we endeavor to align data with releases by splitting data into 1/4/12 partitions, building models capturing the characteristics of certain periods.

Table 4.1: Eclipse 3.x release dates and target reports in our dataset showing major and service releases from 2004-2011 for version 3.0-3.6.

Version	Release date			#Bug reports
	major release	1st Service release	2nd Service release	
3.0	June 25, 2004	September 16, 2004	March 11, 2005	3,177
3.1	June 17, 2005	September 28, 2005	January 25, 2006	2,699
3.2	June 29, 2006	September 28, 2006	March 1, 2007	2,548
3.3	June 28, 2007	September 28, 2007	February 21, 2008	1,990
3.4	June 25, 2008	September 24, 2008	February 25, 2009	1,813
3.5	June 24, 2009	September 25, 2009	February 26, 2010	1,490
3.6	June 23, 2010	September 24, 2010	February 25, 2011	994

4.3 Study Design

This section involves a presentation of the design of our case study intended to address the three research questions.

4.3.1 Overview

The primary objective of our case study is to understand the impact of release cycles on the bug fixing process for the specific case of bug priority prediction. In research question 1, we investigate metrics on the fixing activities for each month, comparing them to analyze the extent of variation in bug-fixing activities during the release cycle. After analyzing the impact of the release cycle on the bug-fixing process, we evaluate the impact of the release cycle on the priority prediction in research questions 2 and 3. For these two research questions, we build four cycle-aware models (CYC) involving different data sizes and approaches, considering the difference in characteristics of various periods within the 1-year release cycle. We then compare these with the opaque model built with the full data that neglects the release cycle.

Table 4.2: The number of bugs by priority in our datasets.

Priority	# bug reports	
P1	223	1145
P2	922	
P3	13395	13395
P4	79	171
P5	92	
sum	14711	

4.3.2 Dataset

Summarized datasets for this study are presented in Tables 4.1 and 4.2. We selected the Eclipse platform 3.x project because it involves development over a long duration, providing enough data for investigating if the bug-fixing process is affected by the release cycle. Also, the Eclipse 3.x products involve a simple and periodic release. The three releases annually comprise a main release in June and two service releases in September (SR1) and around February (SR2). Such a simple and periodic release cycle facilitates understanding the results associated with our research questions. Although data from Eclipse 4.x is applicable, these were avoided because these are at a turning point.

In this study, we include data from the release date of Eclipse 3.0 to that of Eclipse 3.7, meaning data for Eclipse 3.7 and 3.8 are excluded. This is because Eclipse 3.8 is for bug-fixing and JAVA7 support ², which is different from the other version. In this study, we separate the Eclipse platform 3.x project data into year and month, aggregating them on a monthly basis (i.e., separated 12 datasets corresponding to 12 months). Since the datasets are partitioned by year (instead of release version), these datasets lack the version information. Figure 4.2 illustrates the procedure for creating the monthly datasets.

Since projects frequently receive invalid reports or duplicated reports [88] that may produce a bias during analysis, we employed only bug data tagged “FIXED” in the resolution field of Bugzilla. Moreover, we use syntactic analysis [89] to specify commits for

²Eclipse3.8: https://www.eclipse.org/projects/project-plan.php?planurl=http://www.eclipse.org/eclipse/development/plans/eclipse-project_plan_4.2.xml, Last Accessed: January 2020

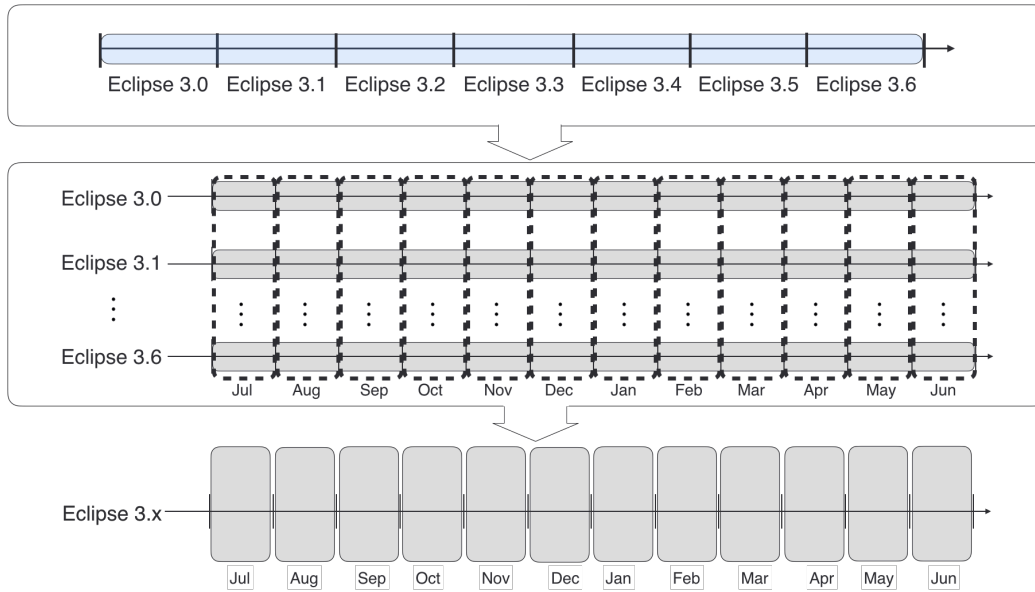


Figure 4.2: Illustration of the procedure for creating our datasets. The data are separated by version and merged by month into 12 datasets. The developers switched to a rapid release cycle from Eclipse 4.6 after the first release (Eclipse 4.2).

each bug; otherwise, we removed the bugs fixed by unidentified commits. When handling descriptions (free text data) in the bug reports for measuring metrics, we eliminated source code and stack trace in the descriptions, as these might lower the performance of the predictions [90].

4.3.3 Metrics

Tables 4.3 and 4.4 show the list of metrics used in this study. We collected data from a bug tracking system and code versioning system, and then measured metrics from the data. The metrics are classified into two types (bug unit and daily unit). Metrics in Bug Unit are measured for each score associated with the bug report (e.g., the number of files changed for the bug) to capture bugs that are fixed. Metrics in the Daily Unit are measured by counting the bug-fixing activities (e.g., the number of files committed per day) per day to capture bugs that are fixed or how these were fixed.

In the Daily Unit metrics, we utilize *# High/Mid/Low priority reports* and *# High/Mid/Low priority closes*. We transform priorities (P1, P2, P3, P4, P5) into three grades (High, Mid, Low) because there are no obvious thresholds existing between P1 and P2/P4 and P5. Therefore, we maintain the changes from the default priority (P3) and merge higher (P1 or P2) and lower (P4 or P5) priorities. Similarly, for the *# High/Mid/Low/Enh severity reports* and *# High/Mid/Low/Enh severity closes*, we replace the blocker, critical, and major into High, normal into Mid, and minor or trivial into Low. Here, since a report whose severity is enhancement is not a bug, we make it independent from the High/Mid/Low and introduce another grade “Enh”.

Table 4.3: Details of measured metrics (Daily Unit) employed in this study showing reports, closes, changes, comments, and commits.

unit	dim.	metrics	description
Daily	Report	# <i>[High / Mid / Low] priority reports</i>	Number of bugs which will receive [High / Mid / Low] priority reported within the day
		# <i>[High / Mid / Low / Enh] severity reports</i>	Number of bugs which will receive [High / Mid / Low / Enh] severity reported within the day
	Close	# <i>[High / Mid / Low] priority closes</i>	Number of bugs whose priority is [High / Mid / Low] priority closed within the day
		# <i>[High / Mid / Low / Enh] severity closes</i>	Number of bugs whose priority is [High / Mid / Low / Enh] severity closed within the day
	Change	# <i>priority changes to [High / Mid / Low]</i>	Number of changes the priorities from any priority to [High / Mid / Low] priority
		# <i>severity changes to [High / Mid / Low]</i>	Number of changes the severities from any priority to [High / Mid / Low] severity
		# <i>priority changes going [up / down]</i>	Number of changes the priorities from any priority to [higher / lower] priority
		# <i>severity changes going [up / down]</i>	Number of changes the severities from any severity to [higher / lower] severity
	Comment	# <i>comments per day</i>	Number of comments which developers discuss in bug reports within the day
		# <i>people commenting per day</i>	Number of developers who discuss in bug reports within the day
	Commit	# <i>commits per day</i>	Number of commits whose commit log include the bug (ID) used in this study
		# <i>changed files per day</i>	Total number of files changed by the commits specified in # commits
		# <i>[added / deleted] rows per day</i>	Total number of rows [added / deleted] by the commits specified in # commits

Table 4.4: Measured metrics (Bug Unit) used in this study showing reports, changes, comments, fixes, and commits.

unit	dim.	metrics	description
Bug	Report	<i># description words severity</i> <i>is enhancement is specified version Bayesian score</i>	Number of words in the description of the bugs Scale given by the bug's severity, from 0 (enhancement) to 6 (blocker) 0 or 1 if the bug whose severity is enhancement 0 or 1 if versions effected by the bug are specified Probability which will receive high priority, produced by the Bayes classifier built with the description in the historical bug reports
	Change	<i># status changes</i> <i># assigns</i> <i># CC's</i>	Number of status changes occurred until the bug are fixed Number of assigned developers until the bug are fixed Number of developers who want to know the progress is changed
	Comment	<i># comments per bug</i> <i># people commenting per bug</i> <i>median of comment words</i>	Number of comments on the bug report Number of developers who commented on the bug report a median value of words in a comment
	Fix	<i>assigning-time</i> <i>fixing-time</i>	Days from the bug are reported to assigned Days from the bug are assigned to fixed
	Commit	<i># commits per bug</i> <i># changed files per bug</i> <i># [added / deleted] rows per bug</i> <i>median of changed files by commit</i> <i>median of [added / deleted] rows by commits</i> <i>median of [added / deleted] rows by files</i>	Number of commits needed to fix the bug Number of files changed to fix the bug Number of rows [added / deleted] to fix the bug Median value of changed files by commits Median value of [added / deleted] rows by commits Median value of [added / deleted] rows by files

4.3.4 Prediction models

In the RQ2 and RQ3, the models are intended to predict whether a bug receives a high priority (P1 or P2) or does not. We explain the procedure for creating the prediction models and then evaluate them. To confirm the effect of the release cycle, we compare models trained with data for different periods. We build cycle-aware models (CYC) trained with data for a specific period and an opaque model (OPA) trained with the entire data.

Model Construction. A summary of the procedure for building the models is displayed in Figure 4.3. The procedure comprises the following five phases: preparation, creating datasets, model building, inference, and prediction.

1 Preparation phase

We measure all metrics using Eclipse platform data for the entire period. To input the daily activity metrics into the prediction models, we refer to the daily activity metrics of the day prior to the reported date of each bug and use them as the metrics of the bugs. Thereafter, we filter all Eclipse data to the Eclipse platform 3.x data, and then split the complete data set into N-split data sets for the CYC, based on the bugs reported month.

2 Making datasets phase

The N-split datasets are further split into training data (90%) and testing data (10%). Then, we copy the training data of the CYCs and combine them as training data for the OPA.

3 Model building phase

We build four models including Bayes, LDA, Impute and Prediction.

I. Bayes model

We use the Naive Bayes classifier to compute the “*Bayesian score*”, which is a probability that will receive high priority in the metrics list (Table 4.3). The probability of a bug report b belonging in class c (high priority) is computed

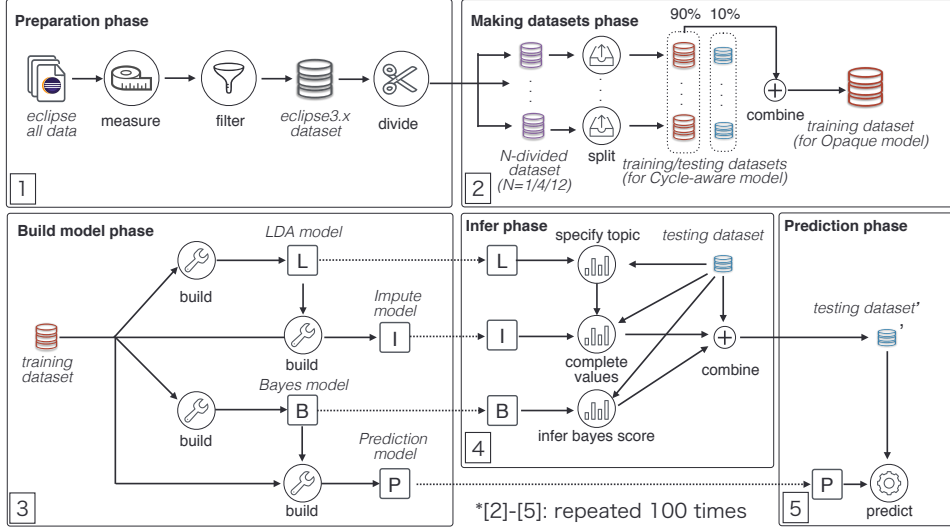


Figure 4.3: Summary of the procedure for completing unknown metrics when new bugs are reported. The new bugs are provided through the inferred metrics by the LDA and Impute models.

as below.

$$P(c|b) \propto P(c) \prod_{1 < k < n_b} P(t_k|c) \quad (4.1)$$

where $P(t_k|c)$ represents the conditional probability of the term t_k occurring in a bug report of class c . $P(c)$ shows the prior probability of a bug report belonging class c , and n_b is the number of terms in bug report b .

To calculate the probability, we consider each text of the bug reports and the priority label, so that the classifier learns relevant information in the text of high priority bugs. The texts are preprocessed via tokenization, lemmatization, and stopwords removal. Finally, the texts are vectorized and weighted using TF-IDF. When a new bug is reported, the classifier takes an input of the description in the reported bug and can compute the possibility that the bug will receive high priority.

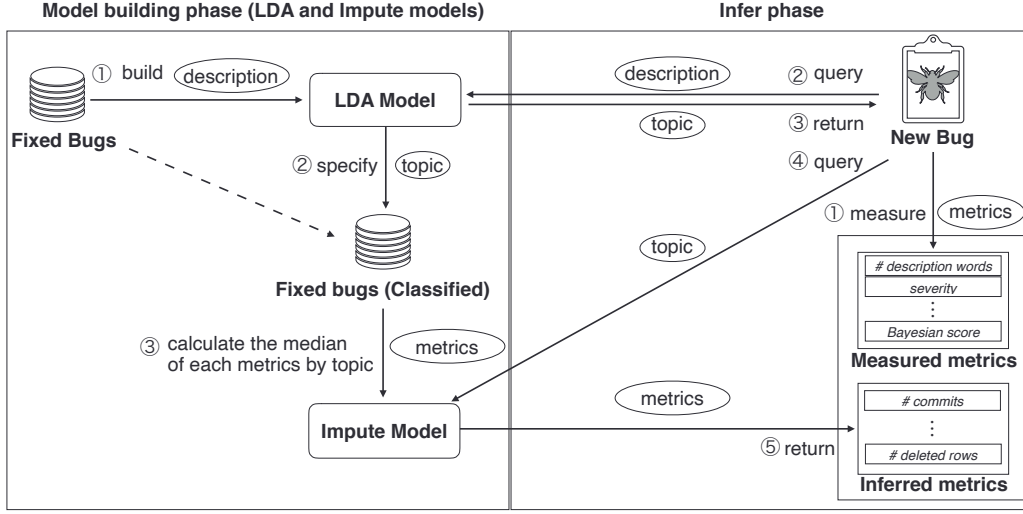


Figure 4.4: The procedure of completing unknown metrics when new bugs are reported. The new bugs are provided with the inferred metrics by LDA and Impute models.

As for computing all metrics (not only the “*Bayesian score*”), the OPA and CYC use each dataset, that is, the OPA uses a full dataset and CYCs utilize the split datasets. However, the performance of the Bayes model is known to significantly rely on the amount of data [36]. Hence, we create another type of CYC, which is trained with full data to compute the “*Bayesian score*”. To distinguish the new CYC and the simple CYC, we name the new CYC as hybrid CYC (H-CYC), while the simple CYC is termed pure CYC (P-CYC), with the “*Bayesian score*” by using divided datasets. In other words, the H-CYC shares the same “*Bayesian score*” computed using the divided datasets. Alternatively, the H-CYC shares the same “*Bayesian score*” as the OPA, but shares the other metrics with the P-CYC.

II / III. LDA models and Impute models

When new bugs are reported and their priorities require prediction, metrics in the Bug Unit (except for the report dimension) are unknown such as the “#

status changes” and “# comments per bug”. This is because these metrics are able to be measured after the bugs are fixed. Likewise, for the testing datasets in this experiment, the original metrics in the bug reports were not utilized. We infer the metrics from similar bugs using the LDA and Impute models. The procedure for building the LDA and Impute models for inferring the unknown values of the metrics are outlined in the left side of Figure 4.4.

First, we build the LDA models with the training datasets to classify the bugs in the training datasets into clusters (topics). To classify the bugs, the LDA model sets the number of topics in advance. Referring to previous studies [91, 92], we set the number of topics by dividing the number of bugs by 2.5. Then, for each topic, we calculate the median of each metric and store the medians in a model (named the Impute model).

Furthermore, each bug in the testing datasets is fed into the LDA models trained with the training datasets and the most relevant topic of each bug is inferred. Finally, the missing metrics are determined by the Impute model, referring to the median values of the metrics according to the inferred topic. Here, the LDA and Impute models are trained for each training dataset.

IV. Priority prediction models

Using the random forest algorithm, we build classifiers to enable the classification of bugs as high priority and non-high priority. This is a classification algorithm that applies multiple decision trees as weak learners. The algorithm outperforms basic classification algorithms in terms of its potential to measure the importance of prediction variables. To build decision trees, we employ the Gini Impurity that is a measurement of the likelihood of an incorrect classification of a new instance of a random variable. It has been calculated as

$$G = \sum_{i \in C} p(i) * (1 - p(i)) \quad (4.2)$$

where $p(i)$ is the probability of picking an item assigned to class i .

When building the priority classification models with the measured metrics

in the training datasets, to improve the performance of the classifications, we remove the metrics with a variance inflation factor (VIF) above 5. The VIF helps detect multicollinearity that lowers prediction accuracy and it is formulated as $1/(1 - R^2)$, where R^2 is the coefficient of determination obtained by regressing the target metrics on the remaining metrics.

Furthermore, we employ a feature selection via the information gain and ranker search method to remove ineffective attributions. In this study, we set the threshold of the information gain as 0.00 and remove the metrics with information gains below or equal to the threshold.

Next, we discretize the metrics (except for 0-1 variables “*is_enhancement*” and “*is_specified_version*”) into 3-bins based on the frequency, involving high, middle, or low, to improve understanding of the data. We also execute under-sampling on the training datasets to prevent over-fitting. Finally, after building the classifiers, we input the testing datasets in the prediction model to classify bugs with high priority.

4 Infer phase

We supplement the missing values in the test data for the metrics that were not measured when the bug was reported (e.g., *#commits per bug*, *# changed files per bug*). The procedure for inferring the unknown value of the metrics is outlined in the right side of Figure 4.4.

Initially, for each LDA model created in the model building phase, each text of the bug reported in the test data is provided as input to infer the topic. We introduce the topic to the Impute model built in the previous procedure (II/III. LDA models and Impute models). The Impute model provides the bug reports with the inferred value, which is the median value for the metrics calculated in bug reports involving the topic in the training dataset. Also, to obtain the “*Bayesian score*” for test data, each text data of the reports is introduced to the Bayes model created in the model building phase to infer the score.

[5] Prediction phase

Finally, we provide priority prediction models with the complemented test data. The predict models compute the probability that the bug report b receives a high priority and select the more likely class (high priority or not) as follows:

$$c = \arg \max_{c \in C} p(b_i) \quad (4.3)$$

Evaluation. To evaluate the performance of the CYC and OPA, we employ the recall, g-mean, and ROC-AUC as performance measures. The recall and g-mean are represented by a confusion matrix of the actual and predicted labels in Table 4.5, while the ROC-AUC is simply obtained from the area under the ROC curve. The ROC curve is plotted with the false-positive rate (equals $1 - \text{specificity}$) as the x-axis and the true positive rate (equals recall) as the y-axis for each classification thresholds in descending order of positive probability. Although many studies employ precision and f-score in addition to recall and the AUC-ROC, we use recall, g-mean, and ROC-AUC because the ratio of the positive and negative in our dataset biases to negative labels (not high priority bugs), and precision is known to be sensitive to the imbalance ratio of the test set [93]³. Therefore, the precision and f-score (calculated with precision) are not appropriate in this study. Instead, we employ the g-mean, capable of equally evaluating negative and positive because it is calculated based on the ratio of the positive and negative labels.

Note that the ROC-AUC also focuses on negative labels and some studies suggest using the PRC-AUC instead when the dataset is imbalanced [94, 95]. The PRC-AUC is calculated like the ROC-AUC but utilizes the Precision-Recall curve instead of the ROC curve. The Precision-Recall curve is plotted with the recall on the x-axis and the precision on the y-axis for each classification thresholds in descending order of positive probability. The PRC-AUC is less affected by numerous negative labels because the Precision-Recall curve does not use False Negative for its calculation. However, the recall relatively focuses on the positive label and employing the PRC-AUC can produce a biased evaluation for the positive label. To prevent emphasizing one class over another, we still use the ROC-AUC in addition to the recall and g-mean.

³We show precision in tables and figures but we do not compare

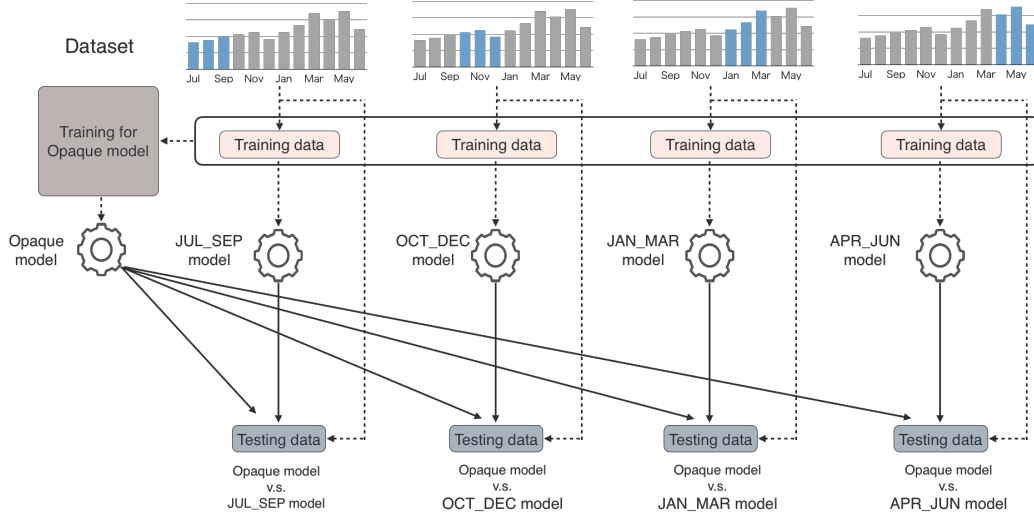


Figure 4.5: An evaluation example for the 4-split models. The cycle-aware models trained with quarterly training datasets predict priorities for testing datasets of the same quarter. The opaque model predicts priorities for the four quarterly testing datasets.

To ensure the stability of our results, we use the holdout verification approach, involving using 90% of data for training and 10% for testing, computing the median of the recall, g-mean and ROC-AUC (abbr. AUC) for 100 times repetition of the verification. However, if we apply the OPA to the testing data by simply dividing the whole dataset by 9:1, the performance is not equally evaluated, which is not a representative score for each period. Because the number of reported bugs in each month is skewed (especially, from March to June), which is close to the score for the period bugs are reported the most, and far from the results of the others. To compare the CYCs with OPA, after dividing the entire dataset into training and testing, we reuse the training data for each CYC and combine them for building the OPA. For each period of the testing datasets, we predict the priority labels with the OPA, and this is compared with the result predicted by the CYC corresponding to the period. An example of the evaluation method for 4-divided models is depicted in Figure 4.5.

Table 4.5: Confusion matrix for a two-class problem and performance measures

		Predicted	
		Positive	Negative
Actual	Positive	True Positive (TP)	False Negative (FN)
	Negative	False Positive (FP)	True Negative (TN)

$$Precision = \frac{TP}{TP + FP} \quad (4.4)$$

$$Recall = \frac{TP}{TP + FN} \quad (4.5)$$

$$Specificity = \frac{TN}{FP + TN} \quad (4.6)$$

$$G-mean = \sqrt{Recall \times Specificity} \quad (4.7)$$

4.4 Study Results

We present the motivation, approach and results of the three research questions.

RQ1: Do defect management activities vary within the release cycles?

Motivation. Most projects involve release cycles, and even in the case of one cycle, the aims of releases vary. According to the aims, developers switch what they focus on, like either implementing new features or fixing defects. Furthermore, depend on the focus, the characteristics of the produced data also vary. Disregarding the characteristics for specific periods render the quality of data-driven tools lower.

However, most studies on defect management assume that the characteristics for all periods are identical across the release cycle. To the best of our knowledge, no studies investigating the impact of the release cycle on the characteristics of data exists. As RQ1,

we therefore, investigate if defect management activities and their characteristics vary across the release cycle, by measuring the activities or comparing various metrics during specific periods.

Approach. For this research question, we initially visualize the bug-fixing activities during the release cycle. We measure the number of bugs reported and fixed and the reporting and fixing times. Then, we apply statistical tests to the metrics in Tables 4.3 and 4.4^{4, 5} to clarify differences in each period. We use the Kruskal-Wallis test to evaluate whether statistical differences exist between the metrics for each month. Since the Kruskal-Wallis test can only detect if differences exist among the distributions, we apply the Mann-Whitney U-test with a Bonferroni correction to all pairs of distributions, as a post-hoc test for evaluating pairs of distributions with statistical differences. Finally, we measure the effect-size ($r = Z \text{ score} / \sqrt{\# \text{ of samples}}$) [96] to confirm larger or smaller distributions, and then remove the results with effect-size below 0.1. Here, to make the results understandable, we divided the data into four groups (aggregated quarterly from July immediately after the main releases) before applying the statistical test. The data from July to September will be referred to as JUL_SEP, that from October to December as OCT_DEC, whereas that from January to March is JAN_MAR, and that from April to June as APR_JUN.

Finding 1-1. The number of reports growing towards the end of the cycle. Figures 4.6 and 4.7 display the number of bug reports and bug fixes for each month, respectively. From July to November, the medians of the bug reports (except for August) and fixes steadily increase until the activities temporarily attain a low in December. In the new year, the number of bug reports and the fixes rise again from January to April. The numbers of reports and fixes are higher from January to May than at any other time during the first half of the release cycle. These significantly decrease in June, the month of the main release.

⁴We do not apply the test to 0-1 variables (*is_enhancement*, *is_specified_version*).

⁵If the bugs are unassigned, measuring the *assigning-time* and *fixing-time* is not possible. Therefore, we regard the bugs as assigned at the time of reporting, setting the assigning-time and fixing-time as 0, and the days from reported to fixed, respectively.

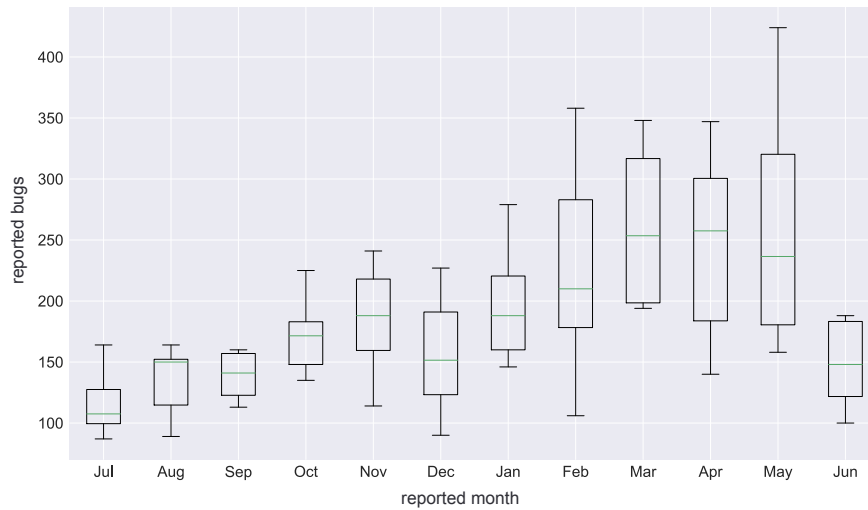


Figure 4.6: Box plot showing the number of bugs **reported** in each month

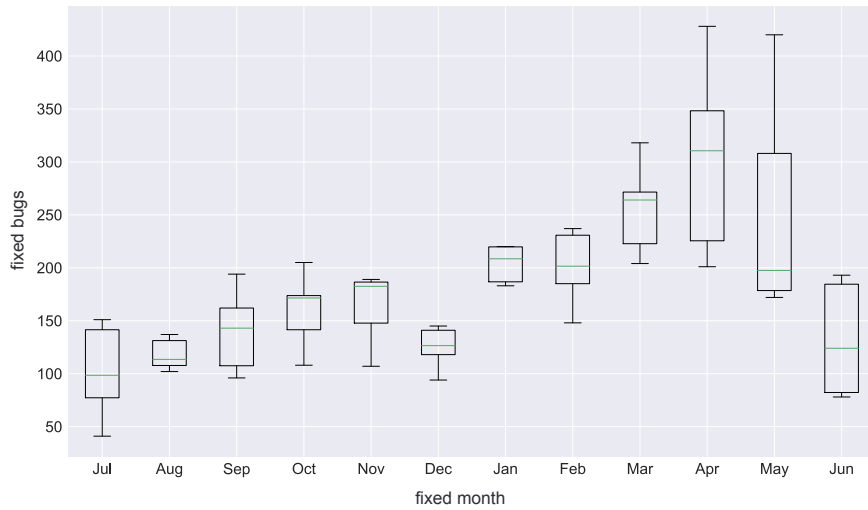


Figure 4.7: Box plot showing the number of bugs **fixed** in each month

The data demonstrates that the numbers of reports and fixes vary within the release cycle, and increase towards month for the main releases. The numbers of bugs reported and fixed during the peak period are almost twice and three times the number of the least frequent period, respectively. This suggests that the prediction model involving the entire data relies on the data at the end of the release cycle. This lowers the performance for the other periods, if the bug-fixing activity varies across the release cycle.

Finding 1-2. The activity of fixing high priority bugs varies more dynamically than the activity of reporting high priority bugs. Figures 4.8 and 4.9 respectively, illustrate the percentages of the reported and fixed high priority (P1 or P2) bugs. For the percentage of reported high priority bugs, the maximum median is 7.5 in May, followed by 7.1 in July (i.e., immediately before and after the main release), while the minimum median is 3.3 in February. Conversely, for the percentage of fixed high priority bugs, the maximum median is 9.9 in May and the minimum median is 2.2 in November. The percentages of reported and fixed high priority bugs attain the maximum in May.

The difference between the maximum and minimum of the fixed high priority bugs are larger than that of reported (reported: 4.2, fixed: 7.7). Also, the variances of the median percentages of fixed bugs is higher than that of bugs reported (reported: 2.2, fixed: 4.0). Hence, fixing high priority bugs activity varies more dynamically than that of the reported high priority bugs activity.

Finding 1-3. Stabilization of the backlog for the next release fixes of high priority bugs. Using the same method as in Figure 4.10, a heat-map filtered by high priority bugs (P1 or P2) is shown in Figure 4.11. For the fixed bugs, in April, vertical alignment of the red shade cells is evident. This indicates that many of the bugs reported between October (the month immediately after the first service release) and March and are fixed in April. That is, in April, while aiming for the main release, developers focus on fixing high priority bugs to stabilize the products.

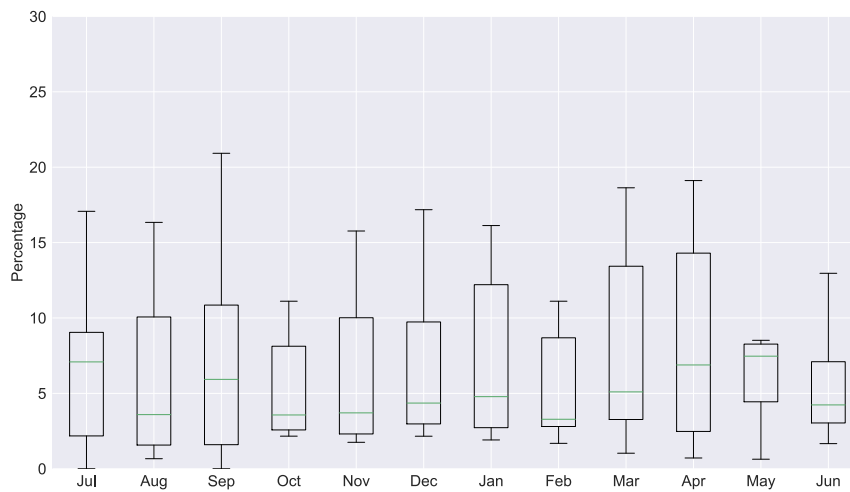


Figure 4.8: The percentage of **reported** high priority bugs in each month

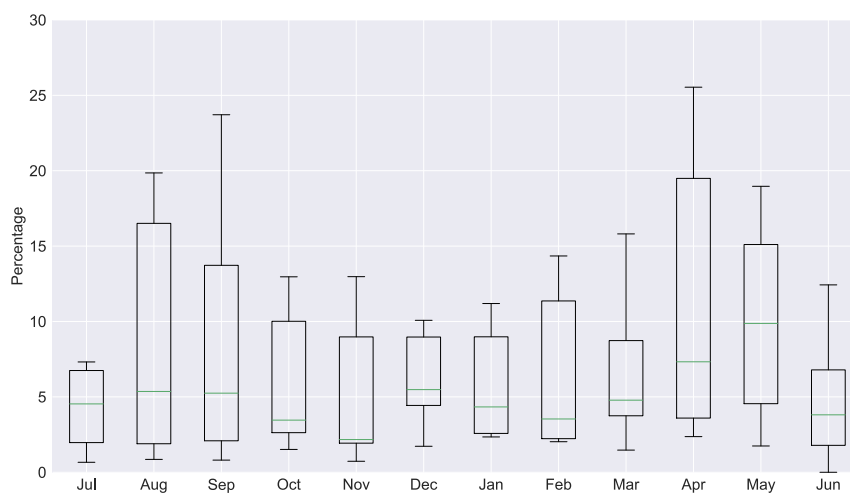


Figure 4.9: The percentage of **fixed** high priority bugs in each month

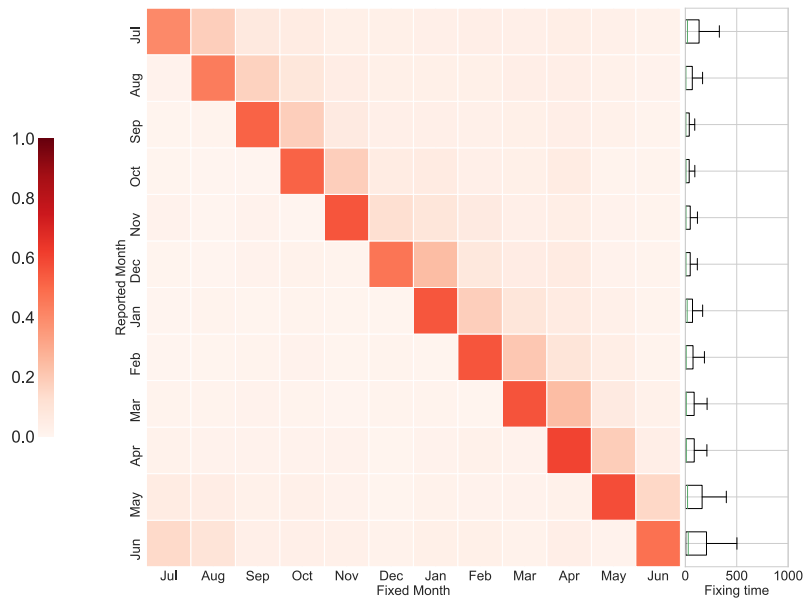


Figure 4.10: Display of percentages of the bugs reported and fixed in each month.

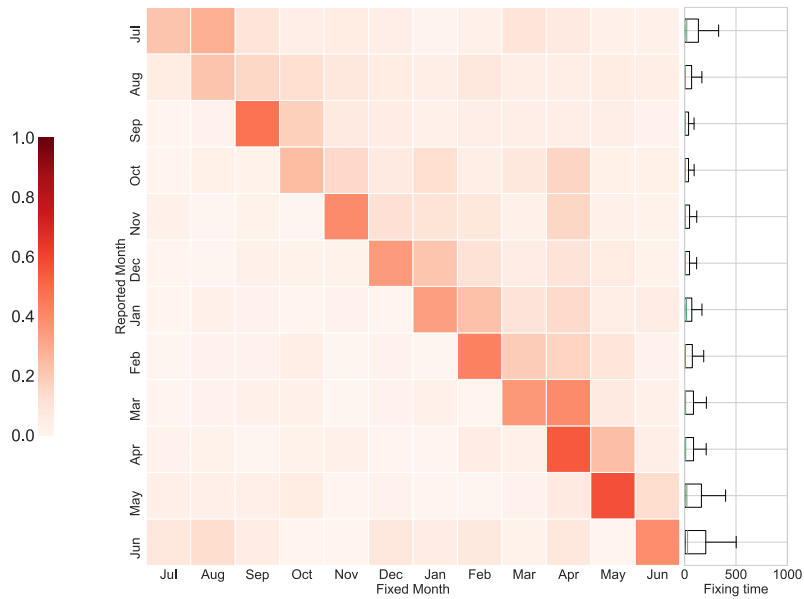


Figure 4.11: Display of the percentages of the **high priority** bugs reported and fixed in each month.

Table 4.6: Effect-size between quarters (+ indicates the distribution is larger than the other quarters, – indicates the distribution is smaller than others)

Unit	Dimension	Metrics	Quarter			
			JUL SEP	OCT DEC	JAN MAR	APR JUN
Daily	Report	# High priority reports	–			+
		# Mid priority reports	–	–	+	+
		# High severity reports	–			+
		# Mid severity reports	–	–	+	+
	Close	# High priority closes	–			+
		# Mid priority closes	–	–	+	+
		# High severity closes	–	–	+	+
		# Mid severity closes	–	–	+	+
	Change	# priority changes to High	–			+
		# severity changes to High	–	–		+
		# priority changes going up	–			+
		# severity changes going up	–	–		+
		# severity changes going down	–			+
	Commit	# commits per day	–		+	+
		# changed files per day	–		+	
		# added rows per day	–		+	
		# deleted rows per day	–		+	
Bug	Report	Bayesian score	–	+	+	+
	Change	# CCs	+		–	
	Comment	# people commenting per bug	+		–	+

Finding 1-4. Developers’ activities vary across quarters. The effect-sizes of at least 0.1 for metrics with a statistically significant difference, as measured by the Kruskal-Wallis and the Mann Whitney tests are presented in Table 4.6. The “+” indicates that the effect-size is larger than those for the other quarters, while the “–” indicates that the effect-size is smaller than those for the other quarters. We note that “+” refers to the largest and “–” refers to the smallest. For example, consider the following form of a bug report: Quarter A <Quarter B <Quarter C <Quarter D. In this case, Quarter B and Quarter C are shown as blank.

In the Kruskal-Wallis and Mann Whitney tests, 45 out of 50 metrics exhibit statistically significant differences. Through the post-hoc and the effect-size filtering, 24 metrics are eliminated, and 21 metrics are finally retained (See Table 4.6). First, comparing

the first half of the year (JUL_SEP and OCT_DEC) and the second half of the year (JAN_MAR and APR_JUN), bug are reported and closed more frequently in the second half before the main release than the first half after the release. Subsequently, we describe the characteristics of developers' bug-fixing activities for each quarter.

JUL_SEP: Most of the metrics in Daily Unit are smaller than those for JAN_MAR or APR_JUN. However, in the Bug Unit, the *# CCs* and *# people commenting per bug* in JUL_SEP are superior to those JAN_MAR. Therefore, developers likely pay attention to bugs reported after the main release. Interestingly, we examined the e-mail addresses of the people commenting, and realized that outside developers (hostnames without "ibm.com") commented to the bug reports more frequently in July (immediately after the release) than in other months (June: 82%, other months: [19%-68%]).

OCT_DEC: Similar to JUL_SEP, most of the metrics in Daily Unit for OCT_DEC involve negative effect-sizes. The characteristic metrics in OCT_DEC, with a positive effect-size is the *"Bayesian_score"*. Overall, we can see developers are less active in OCT_DEC.

JAN_MAR: As described earlier, significantly higher numbers of bugs are reported and fixed in JAN_MAR and APR_JUN. Also, developers often commit to the repository in JAN_MAR and APR_JUN. However, only in JAN_MAR, the *"# changed files per day"*, *"# added rows per day,"* and *"# deleted rows per day"* are highest. Therefore, implementation activities are evidently most common in JAN_MAR.

APR_JUN: During this time, developers fix many bugs similarly to the JAN_MAR period, but they focus on reporting and fixing high priority and severity bugs. Also, they frequently change priority and severity. Moreover, similar to the JAN_MAR period, developers frequently commit, but the *"# changed files per day"*, *"added rows per day,"* and *"deleted rows per day"* are smaller. Thus, considering the main release, developers triage bugs and fix high priority or high severity bugs without considerable changes.

RQ1: Developers' activities vary during the release cycle. Specifically, developers actively discuss bug fixes in JUL_SEP, dynamically change the source code in OCT_DEC, and improve the quality for the main release in APR_JUN.

RQ2: Do cycle-aware models outperform the cycle-unaware models?

Motivation. Based on the results for RQ1, developers’ activities and the characteristics of data vary during release recycle. This suggests that the model trained with all available data during the release cycle might miss subtle differences.

Moreover, the number of bug reporting in the most frequent period is approximately twice as many as those in the least frequent period. Using data derived from the entire release cycle might introduce a bias to the most frequent period. For this research question, we compare the performance of cycle-aware models (CYC), exploiting the differences of activities with the opaque model (OPA) utilizing the maximum available data.

Approach. For this research question, we evaluate if the priority prediction improves by considering the release cycle. The CYCs are built with quarterly split data, starting from July, based on the bug-reporting date. The data set length of three months (quarter) is commonly used in companies, and in Eclipse, each period involves either main release, service release, or December break. Note that, for RQ3, we make each CYC (monthly level) more fine-grained and discuss the extent of splitting the data. The data for every three months starts from July; like in RQ1, these are named JUL_SEP, OCT_DEC, JAN_MAR, and APR_JUN. Then, we compare the CYCs with the OPA built with the combined training data from all CYCs. We also apply the Wilcoxon signed-rank test with continuity correction and a Bonferroni correction to confirm the statistically significant difference for each test data.

The values of some metrics are undetermined or cannot be measured at that time the bug is reported. For example, prediction models cannot use the final value of “*severity*” because “*severity*” (i.e., at that time the bug is fixed) often differs from that at the time the bug is reported. Thus, in the prediction models, we input the value of “*severity*” measured at the time that each bug is reported (i.e., we use the information as far as possible when the bug is reported). Likewise, metrics such as “*fixing-time*” and “*# of comments per bug*”, cannot be measured when the bugs are reported. As described in section 4.3.4, the metrics are inferred from similar bugs by using the LDA models and Impute models.

Table 4.7: Performance of the cycle-aware models and the opaque model. The bold scores represent the highest score among the three models with statistical significance. The scores underlined indicate the highest score among the three models but without statistical significance.

Quarter	Model	recall	G-mean	precision	AUC
JUL_SEP	OPA	0.38	0.58	0.21	0.70
	P-CYC	0.36	0.55	0.16	0.68
	H-CYC	0.48	0.62	0.17	<u>0.71</u>
OCT_DEC	OPA	0.35	0.55	0.17	<u>0.69</u>
	P-CYC	0.35	0.53	0.12	0.65
	H-CYC	0.43	0.59	0.14	0.67
JAN_MAR	OPA	0.40	0.57	0.19	<u>0.70</u>
	P-CYC	0.40	0.58	0.16	0.67
	H-CYC	0.43	0.61	0.18	<u>0.70</u>
APR_JUN	OPA	0.35	0.55	0.17	0.67
	P-CYC	0.39	0.55	0.14	0.64
	H-CYC	0.43	0.57	0.16	<u>0.68</u>

Finding 2-1. Pure cycle-aware models do not outperform the opaque model

The performance of the predictions by the P-CYC, H-CYC, and OPA (as a reference, precision is also shown in the Table) are displayed in Table 4.7. The bold scores represent the highest scores among the three models with statistical significance. The scores underlined indicate the highest score among the three models but without statistical significance.

The P-CYC exhibits similar recall scores to the OPA in JUL_SEP, OCT_DEC, JAN_MAR but a slightly higher score in APR_JUN. The differences are -0.02 (-7.0%), 0.00 (0.0%), 0.00 (0.0%), and $+0.04$ ($+10.3\%$) for JUL_SEP, OCT_DEC, JAN_MAR, and APR_JUN, respectively. A statistically significant difference emerges only for APR_JUN. For the g-mean, the P-CYC does not outperform the OPA for all quarters. The differences are 0.03 (-5.1%), -0.02 (-2.4%), 0.01 ($+0.1\%$), and 0.00 (0.0%). The AUC of the P-CYC is consistently lower than that of the OPA, with statistically significant differences for all quarters. The differences are -0.02 (-2.9%), -0.04 (-6.2%), -0.03 (-4.5%), and -0.03 (-4.7%).

Finding 2-2. Hybrid cycle-aware models outperform the opaque model in terms of recall and g-mean

The recall predicted by the H-CYC is consistently higher than that by the OPA. The differences are $+0.10$ ($+20.8\%$), $+0.08$ (18.6%), $+0.03$ (7.0%), and $+0.08$ ($+18.6\%$) for JUL_SEP, OCT_DEC, JAN_MAR, and APR_JUN, respectively, characterized by statistically significant difference for all quarters. For the g-mean, similar to the recall, the H-CYC outperforms the OPA, with statistically significant differences for all quarters. The differences are $+0.04$ ($+6.5\%$), $+0.04$ ($+6.8\%$), $+0.04$ ($+6.6\%$), and $+0.02$ ($+3.5\%$). The AUC of the H-CYC almost equals that of the OPA and the differences are $+0.01$ ($+1.4\%$), -0.02 (-3.0%), 0.00 (0.0%), and $+0.01$ ($+1.5\%$). No statistically significant difference exists for all quarters.

RQ2: The performance of the hybrid cycle-aware model is better than that of the opaque model for the recall and g-mean.

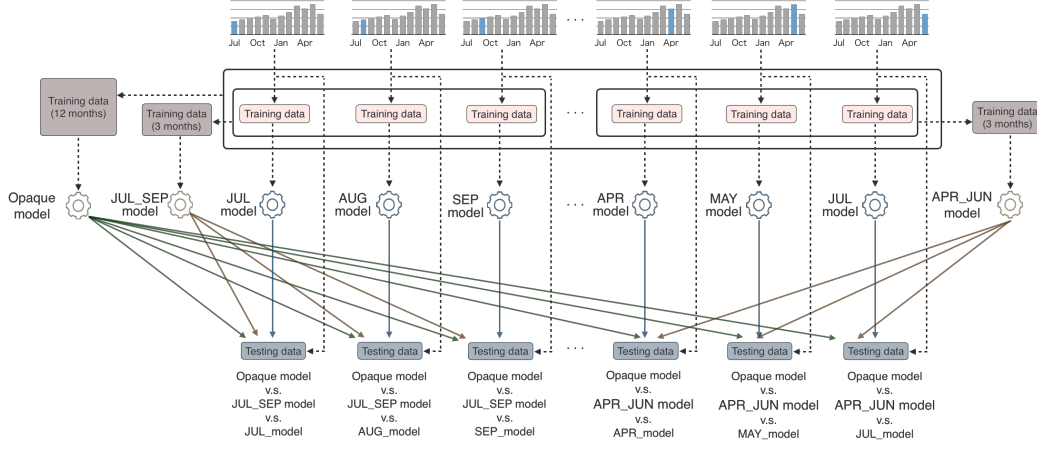


Figure 4.12: An evaluation method for the 1/4/12-split models. The monthly models are trained with monthly training datasets, and predict priorities for testing datasets of the same month. The quarterly models and the opaque model are trained with the training datasets of every three months and all months.

RQ3: What is the right granularity for cycle-aware models?

Motivation. By calculating *Bayesian score* with as much data as possible and using the split data to build the prediction models, the H-CYC models outperform the OPA model. This suggests that the performance of the H-CYC models might be improved through finer-grained data. However, we remain unsure of the fine-grained data extend needed for the H-CYC models.

Approach. We build 12-split models (named Monthly models), 4-split models (Quarter models), and 1-split models (Opaque model). Before comparing the hybrid models, to confirm the validity of the hybrid models for Monthly models, we first compare pure monthly models (P-MoCYC), with *Bayesian score* calculated using data for each month with the hybrid monthly models (H-MoCYC). We then compare the H-MoCYC with the OPA and hybrid quarter models (H-QuCYC).

Figure 4.12 illustrates the evaluation method for RQ3. To build the models, the OPA is created by combining all training data of the 12-split models and H-QuCYC are

made by combining the training data of each quarter for the 12-splitted models. All models predict the priorities of the monthly test data.

Finding 3-1. Hybrid monthly models outperform Pure Monthly models in most of the months

Figure 4.13 shows the performance of the P-MoCYC, H-MoCYC, and OPA. For the recall, although the H-MoCYC exhibits the same score as the P-MoCYC in July, the H-MoCYC frequently outperforms the P-MoCYC from August. The maximum difference is 0.23 in November, with evident statistical significance in the monthly data except for July. The g-mean and AUC of the H-MoCYC are higher than that of the P-MoCYC for all months. For the g-mean, statistically significant data are present monthly, except for July and May. For the AUC, statistical significance appears in the data monthly, except for July and March. Similar to the Quarter models, calculation of the *Bayesian score* requires more data.

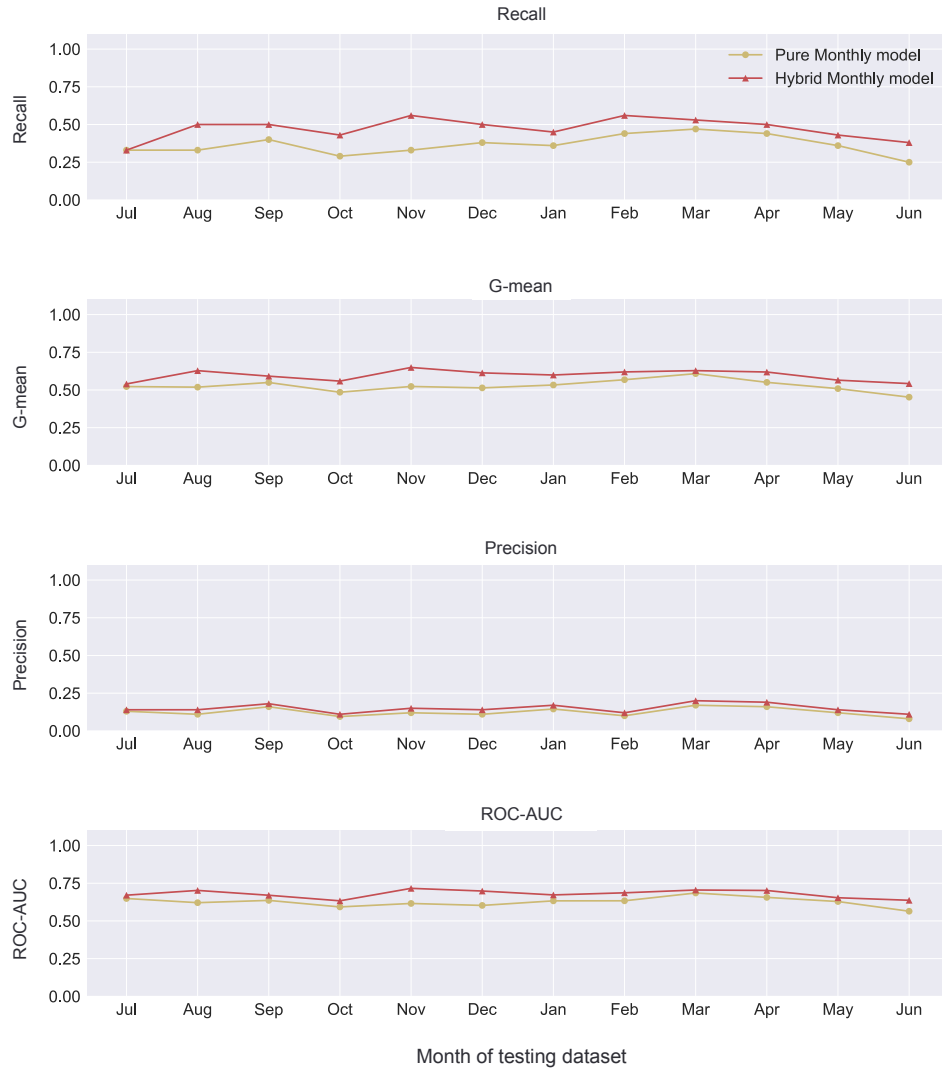


Figure 4.13: Prediction performance of the Pure and Hybrid monthly models highlighting consistent superiority of the hybrid over the pure models.

Finding 3-2. More fine-grained models tend to outperform less fine-grained models

Figure 4.14 shows the median of the predicted performance for 100 iterations each by OPA, H-QuCYC, and H-MoCYC. The average scores from these three methods contact each other in some months for all the performance measures. We plotted the distributions of the scores from these 100 iterations with boxplot and confirmed that the distributions are different. The recall of H-MoCYC is higher than that of OPA in most months, except for Jul and Apr, and it shows statistically significant differences except for Sep and Apr. H-MoCYC outperforms H-QuCYC in the second three months (Oct, Nov, and Dec), Feb, Mar, and May and there are statistically significant differences in Nov, Dec, Feb, Mar, and Jun. The only month in which H-QuCYC outperforms statistically significantly H-MoCYC is Jul.

In terms of the g-mean, the score of H-MoCYC is higher than the score of OPA in Aug, Oct, Nov, Dec, Jan, Feb, Mar, May, and Jun, and statistically significant differences occur in Aug, Oct, Nov, Dec, May, and Jun. Compared with H-QuCYC, the scores of H-MoCYC are lower in the first three months (Jul, Aug, and Sep), and a statistically significant difference occurs only in Jul. In the second three months (Oct, Nov, and Dec), the scores of H-MoCYC are higher, but there are no statistically significant differences. From Jan to Jun, the two models show similar scores with no statistically significant differences.

As measured with AUC, OPA slightly outperforms H-MoCYC in most months, but with statistically significant differences only in July and Sep. In Aug, Nov, and May, H-MoCYC outperforms OPA, but with no statistically significant differences. H-MoCYC outperforms H-QuCYC in Nov, Dec, and May. However, in other months, H-QuCYC outperforms H-MoCYC, and statistically significant differences occur in Jul, Sep, Jan, and Jun.

Overall, H-MoCYC is a better model throughout the year. In particular, for the first three months (from July to September), H-QuCYC is the best model, and for the second three months (from October to December), H-MoCYC is the best model. Note that OPA

did not fit the first half of the year, so it is biased to the second half of the year. **This demonstrates that the developers' activities are not uniform throughout the year.**

In contrast, from Jul to Sep, H-QuCYC provides a good compromise. This is because H-MoCYC may not be trained adequately, due to the smaller number of data points used, resulting in less good performance than for H-QuCYC. Thus, dividing a dataset into a more fine-grained form does not always improve the performance.

RQ3: Hybrid monthly models have better performance than the others over the year. In particular, for the first three months (from Jul to Sep), the hybrid quarterly model is the best.

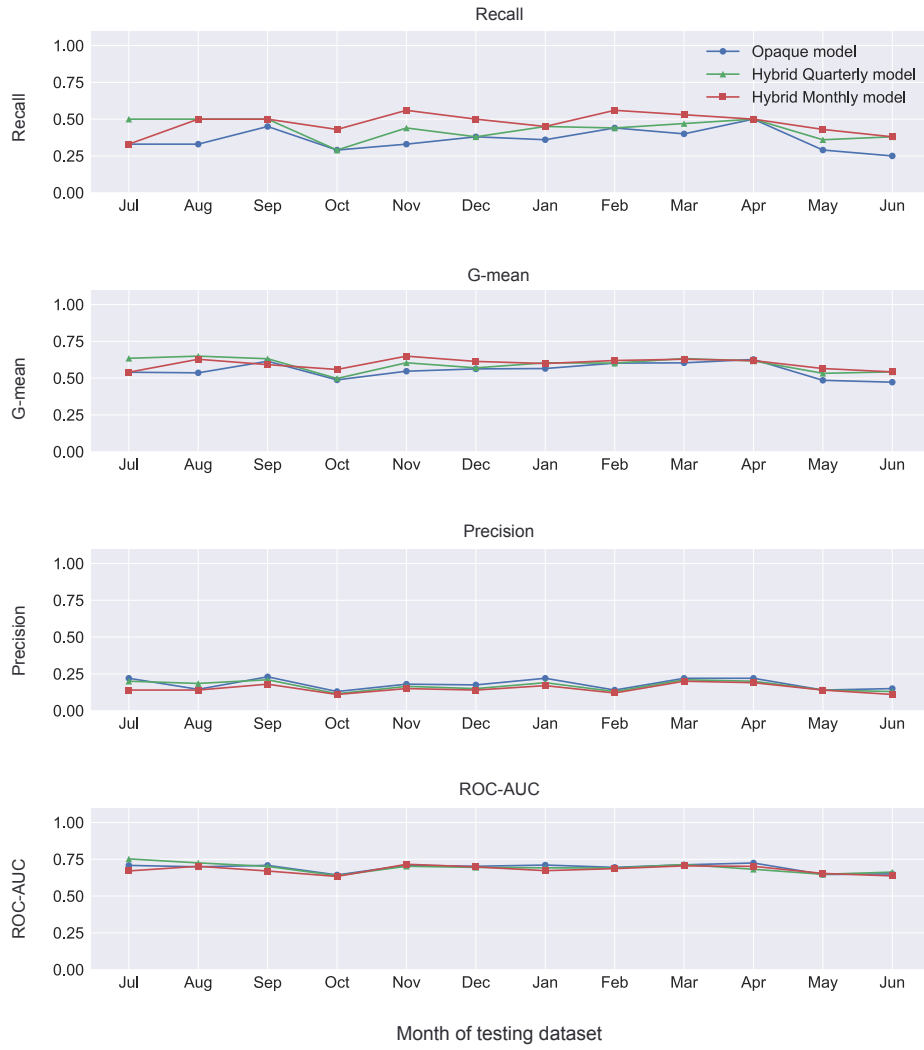


Figure 4.14: The prediction performance of the opaque models, Hybrid quarter models and Hybrid monthly models

4.5 Discussion

4.5.1 Interpretation of Results

In this chapter, we found that each period has different characteristics and that this affects the priority prediction models. However, it is unclear to what extent the difference in the strengths of the CYC and Opaque models depend upon each metric, and how these strength differences affect the performance differences. In this section, we consider the importance of the variables in these prediction models and discuss how splitting the data affects them. The importance of the variables in OPA and H-QuCYC are shown in Table 4.8. Because there are so many variables, we have filtered out the variables with importance less than 0.05 in all models. We determined the threshold 0.05 so that each model accounts for at least 75% of the total importance. Note that, “—” indicates that the metrics have been removed by VIF removal processing or by feature selection. The names of the metrics written in boldface indicate that the size of the effect is larger in RQ1 than in the other quarters (“+” in Table 4.6).

The Magnitude of importance: Bayesian scores are commonly utilized by all models in making predictions, but the magnitudes differ in each model. Interestingly, the OCT_DEC model relies on the *Bayesian score* the most heavily, which corresponds to the findings in RQ1 (“The characteristic metrics in OCT_DEC, where the effect size is positive, is only the *Bayesian score*”). This suggests that the CYC (OCT-DEC) provides greater accuracy than OPA by grasping the characteristics in the period. Similarly, *# High priority reports* is one of the important metrics in JAN_MAR and APR_JUN. However, RQ1 shows that *# priority changes to High* is one of the characteristic metrics in APR_JUN, but this metric is smaller than those in other periods. This might be because *# priority changes to High* is relatively less important in APR_JUN for predicting priority. This suggests that even though the metric is characteristic of the specific period, the release cycle affects the variables less when the metric is less important among all the variables used in the prediction.

Table 4.8: The importance of variables in cycle-aware models and the opaque model (Importance ≥ 0.05). The bolded name of the metric indicates that the effect size is larger in RQ1 than in the other quarters.

Metrics	OPA	JUL_SEP	OCT_DEC	JAN_MAR	APR_JUN
bayesian score	0.31	0.30	0.42	0.31	0.43
# status changes	0.14	0.15	0.12	0.18	0.15
# comments per day	0.07	0.09	0.08	0.07	0.08
# High priority reports	0.07	—	—	0.07	0.03
severity	0.06	0.04	0.04	0.04	0.05
# priority changes to High	0.05	0.04	0.03	0.03	0.02
# comments per bug	0.04	0.05	—	0.02	—
fixing time	0.02	0.05	0.02	0.03	0.02
# description words	0.01	0.04	0.05	0.04	—

The similarity of importance in each model: Figure 4.15 shows the similarity in the variables’ importance in the Quarterly models by hierarchical clustering [97]. In Figure 4.15, the most similar models are OPA and JAN_MAR, which might have resulted in that their recalls in RQ2 are also nearly. Also, this shows that OPA is specialized to JAN_MAR, suggesting that it is not always robust for all periods when building prediction models with as much data as possible. One of the reasons why OPA is a similar model to JAN_MAR is that the amount of the data in JAN_MAR accounts for about 33% of the all data which is the largest percentage. For CYC, the most similar pairs are OCT_DEC and APR_JUN, as compared to JUL_SEP and JAN_MAR. Surprisingly, these are not continuous periods, suggesting that consideration of concept drift over time [77] is always effective in the prediction models. One reason why these are not continuous periods is that it might be involved by the stabilization indicated in RQ1. In JAN_MAR, developers focus on fixing bugs to stabilize the quality of the product. Similarly, in JUL_SEP, they may concentrate on fixing bugs found immediately after the main release. Also, both OCT_DEC and APR_JUN are periods when developers tend to be less active. OCT_DEC

is the period after the 1st service release and APR_JUN is the period after the stabilization.

We show the similarity of the variables' importance for the Monthly models in Figure 4.16. The clusters are roughly classified into four groups, with the months in each group being close to each other. By exploiting these trends, developers might be able to build prediction models to predict priorities even if the projects do not have sufficient data.

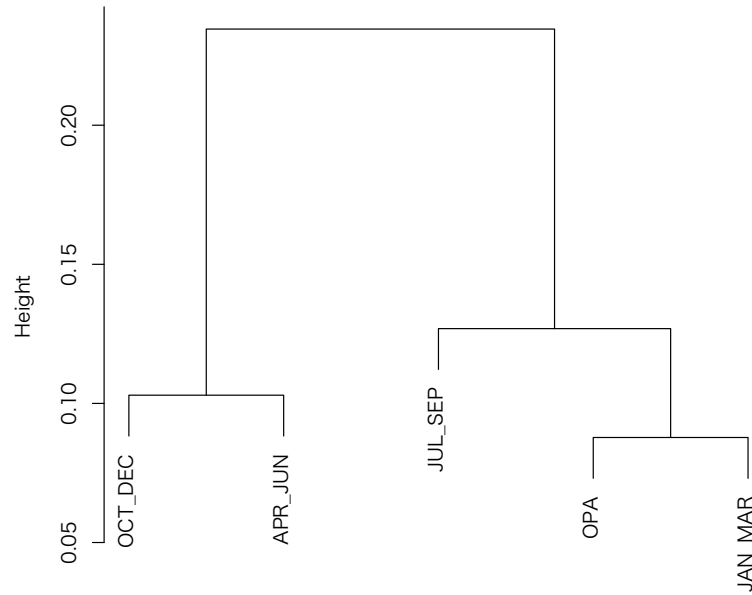


Figure 4.15: The similarity of the variables' importance in OPA and H-QuCYC

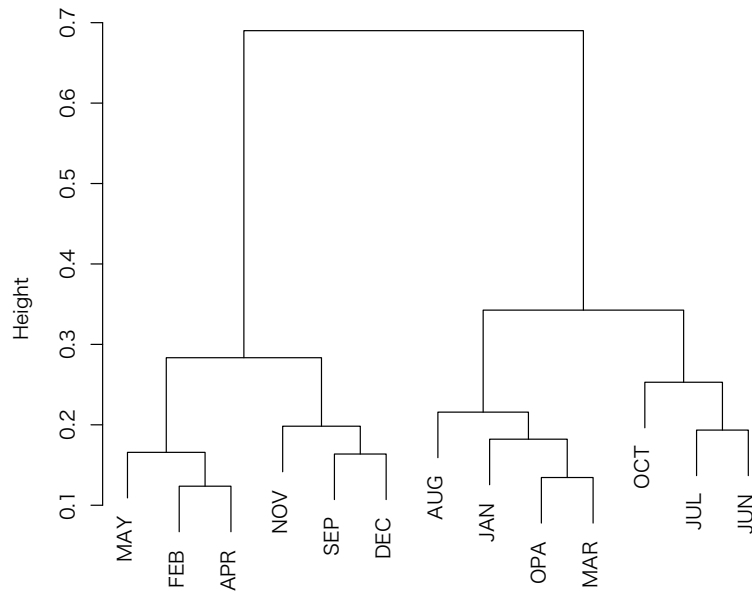


Figure 4.16: The similarity of the variables' importance in OPA and H-MoCYC

4.5.2 Threats to Validity

Internal validity: One concern is that volunteers (especially one-time contributors [98]) may not care about the software release cycle in developing a product. Many volunteer software developers contribute to Open Source Software projects. If a project is supported by one-time contributors for the most part of the development, our findings may not be valid, because such developers may not care about the release cycle in submitting a patch. Thus, the datasets for cycle-aware models should be examined to determine whether each bug is fixed by volunteer developers or by employed developers. Fortunately, in our datasets, one-time contributors account for 0.4% of the work, while 82% of the fixes are provided by developers who have contributed more than 100 times, which does not affect our results much.

The second concern has to do with the division of a dataset into multiple (4/12) datasets. Eclipse has three releases, which occur at the end of June, September, and February, although strictly speaking, the release date is not necessarily the last day of the month. In this study, although we divided the whole dataset into units that extend from the beginning of the month or the quarter to the end of the month or quarter, pre-release bugs and post-release bugs are included. That might affect our results, for example in RQ1, because the fixing time in June are longer than in the other months and because post-release bugs are left until the next release. We can measure this effect precisely by including the post-release bugs into the following month or by making weekly datasets.

The third concern is that we did not take consideration of seasonal variation even though we use time-series data. While seasonality is usually removed in other research areas (e.g., economics), our study has not removed. For example, in December, many developers would take day-off for Christmas and the number of fixes will be dropped at the days. Such temporary drops would not be shown frequently in the year. Thus, we believe that seasonality does not affect tremendously but still, the effect is unsure and then we plan to investigate the seasonality of bug-fixing activity for future work.

External validity: Still another concern is with lack of the generality of our results. Although we studied only the Eclipse Platform project, which employs a periodic and simple release cycle, other projects each have their own release cycles, which are sometimes

complicated. Therefore, the positive effects on the accuracy of the priority classification by cycle-aware models cannot always be provided as promised, and it remains unclear to what extent our results can be extended to other projects. On the other hand, we could first check how many months the period of the test dataset is far from the next release date, and gather data of the period according to the months, (i.g., a month before/after the release). In the future, we plan to evaluate release-cycle-aware models for projects with more complicated release cycles.

A further concern is with the negative effects on the results due to inappropriate managing priority. Saha et al. warned that researchers should pay attention to mislabeling, because most reports concern default severity, and in 65% of those reports the default values are mislabeled [26]. This finding should also apply to priority. Since the bug reports in our dataset also have default priorities, and since they account for 80% of the reports, most bug reports may be labeled with the default value unless labeling is assigned a high priority. If this is correct, a true negative may become a false positive, and a false negative may become a true positive. In that case, however, cycle-aware models are less negatively affected than opaque models, because cycle-aware models make more high-priority recommendations (in other words, more false positives and false negatives) than opaque models. Therefore, we anticipate that cycle-aware models have the possibility of being improved, but opaque models do not.

Construct validity: Finally, we are concerned that we may not have correctly reflected the effects of the release cycle on our results. Bugs fixed before an immediate release are not always included in the release. For example, bug fixes for a maintenance version or for a future major release (e.g., for Eclipse 4.x) might be mixed in our dataset. Since bug reports have a version information tag, if we can utilize them, the effects of the release cycle might be shown more clearly in our results.

4.6 Chapter Summary

Many studies demonstrate unawareness of the existence of releases, alternatively, these inadvertently assume uniform activities within iterations (performing the same task every day) and across iterations (performing the same task in each iteration), which is commonly incorrect. For example, modern software development projects (e.g., agile software development) add new features into products for short periods. Even if in this short period, projects follow the software development process (planning, designing, implementing, and testing). Therefore, the activities at any time during this period varies.

Recently, shortening of the release cycle is common, causing developers to switch development phases to shorter than usual periods. This creates datasets including multiple iterations of the planning, designing, implementing, and testing phases. Prediction models built with such datasets are opaque because these are not aligned with the sequence of events, thereby lowering performances.

We studied the impact of the release cycle alignment on defect management prediction, in particular, bug priority prediction. In this chapter, we proposed release cycle-aware models trained on data within a defined period, considering non-uniform activities within the release cycle. Although the original dataset contained multiple releases, we split the original dataset into several subsets by aligning with major releases and measured the impact of the alignment. Moreover, we compared the prediction performance while changing the datasets granularity to obtain finer-grained analysis/models.

We conducted a case study and showed that developers' activities varies during the release cycle. Based on these findings, we built cycle-aware models for priority prediction. We found that hybrid cycle-aware models outperform the opaque models using full data. Finally, we demonstrated that finer-grained models exhibit higher accuracies.

Chapter 5

Release-Aware and Prioritized Bug-fixing Task Assignments

5.1 Introduction

Most of the proposed task assignment methods have aimed to reduce reassignments by recommending developers who can reliably and quickly fix individual newly-reported bugs, based on previously-reported bugs and their bug-fixing history. However, they possibly concentrate their assignments on a small number of particular developers because the number of past bug fixes differs depending on the developer which makes the training data for each developer imbalanced. Since software development is generally tied to release dates, the number of bugs that can be fixed by even experienced developers before each release is limited. Therefore, the concentration on the specific developers may reduce the number of bugs that the developers can fix by the next release date.

In this research, we propose the Release-Aware and Prioritized Task-assignment Optimization fRamework (RAPTOR) for the test phase, which considers the bug-fixing loads placed on developers, to increase the number of bug-fixes by the next release date. We regard the bug assignment problem as a combination problem between bugs and developers and we formulate it as a multiple knapsack problem to find the optimal combinations. We optimize the assignment process by finding bug assignments that satisfy

certain constraints, aiming to (1) mitigate the task concentration problem caused by existing methods, (2) assign the appropriate amount of bugs to fix more bugs by the next release date.

Chapter Organization: The rest of this chapter is organized as follows. Section 2 describes problems with existing bug triage methods and our key idea for addressing them. Our method is introduced in Section 3, and its implementation is described in Section 4. Section 5 and 6 present our experiments and results, respectively. We discuss the results in Section 7. Finally, Section 8 concludes this chapter.

5.2 Multiple Knapsack Problem

The multiple knapsack problem [99, 100] is an optimization problem that involves finding the best combinations of items (with certain weights and values) to put in a series of knapsacks. Here, each knapsack has a maximum weight that it can carry. Fig. 5.1 gives an overview of the multiple knapsack problem, which extends the well-known knapsack problem to multiple knapsacks. In addition to deciding whether or not to put an item in the knapsack, it requires us to decide what items to put into each knapsack, significantly increasing the computation required. The multiple knapsack problem can be formulated as follows.

$$\text{Maximize : } \sum_{i=1}^m \sum_{j=1}^n v_j x_{ij} \quad (5.1)$$

$$\text{Subject to : } \sum_{j=1}^n w_j x_{ij} \leq c_i \quad (i = 1, 2, \dots, m) \quad (5.2)$$

$$\sum_{i=1}^m x_{ij} \leq 1 \quad (j = 1, 2, \dots, n) \quad (5.3)$$

$$x_{ij} \in \{0, 1\} \quad (j = 1, 2, \dots, n) \quad (5.4)$$

Here, v_j and w_j represent the value and weight of the j -th item, respectively, whereas x_{ij} is the objective variable, representing whether (1) or not (0) to put the j -th item into the i -th knapsack. Expression (5.1) is the objective function and is used to determine whether one combination of objective variable values is better than the other and in this case aims to maximize the total value of the selected items. In contrast, Expression (5.2) is a constraint that denotes that the total weight placed in the i -th knapsack must be less than the maximum weight it can carry (c_i), and Expression (5.3) prevents any item being placed in more than one knapsack. Expression (5.4) denotes the constraint that the x_{ij} should only take values of 0 (not selected) or 1 (selected), i.e., should represent whether the i -th knapsack contains item j .

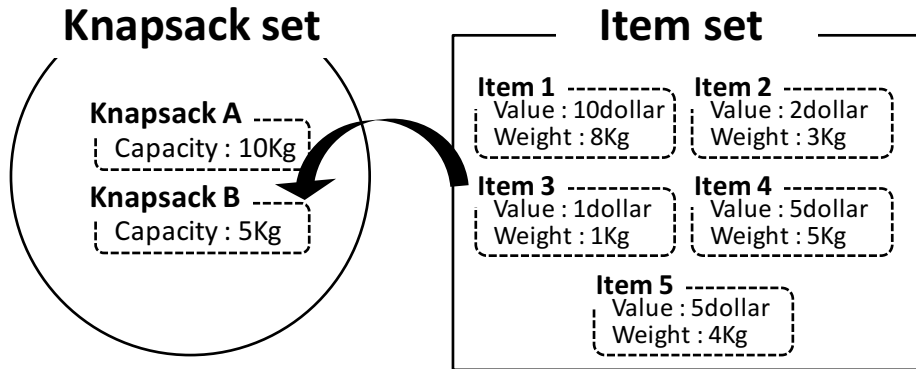


Figure 5.1: Overview of the multiple knapsack problem. The best combination, where the total value of items included in all the knapsacks is the largest, should be determined while the total weight of each knapsack is less than or equal to the capacities in all knapsacks.

The purpose of the multiple knapsack problem is to find combinations of x_{ij} values that maximize the value of Expression (5.1) under the constraints Expressions (5.2), (5.3), and (5.4), which can be reduced easily with a solver such as `lp_solve`¹.

¹Lpsolve: <http://lpsolve.sourceforge.net/5.5/>, Last Accessed: January 2020

Table 5.1: List of terms used in this chapter

Term	Variable	Meaning
Category	k	Bug category (classified by LDA).
Preference	P_{ij}	Used to prioritize developers when assigning bug-fixing tasks. P_{ij} is the probability that developer D_i is the most appropriate for fixing bug B_j .
Cost	C_{ij}	Time taken by developer D_i to fix bug B_j , equal to the median time required by developer D_i to fix a bug in category k in the past.
Limitation	L	Prevents task concentration.
Available assignment time	T_i	Time available for bug fixing within a given period. T_i is the amount of time developer D_i has available: $T_i = \text{Limitation} L - \sum_{j=1}^n C_{ij} * x_{ij}$

5.3 Application of the Multiple Knapsack Problem to Bug Triage

In this chapter, we formulate bug triage as a multiple knapsack problem and use its solution to optimize task assignments. We obtain a combination of items (bugs) and knapsacks (developers) that maximizes the objective (bug-fixing efficiency for the whole project) under each knapsack's weight constraint (maximum time available to each developer or **limit**). The weights are the costs of fixing the bugs (**cost**), and the values are the developers' suitabilities for each bug (**preference**). The terms used in this chapter are summarized in Table 5.1.

Notably, the developers' suitability (preferences) and costs will differ depending on which developers are assigned to which bugs. As a result, the variables in this problem are different from those in the general multiple knapsack problem (we have switched from v_j to P_{ij} and from w_j to C_{ij}).

Preferences (developer suitabilities)

Here, the coefficients for the objective variables in the multiple knapsack problem's objective function are the **preference** P , indicating which developers should be preferred for and can fix particular bug-fixing tasks. The preference of developer D_i for fixing bug B_j is defined as the probability P_{ij} that developer D_i is the most appropriate for the task among all developers (i.e., the total of probability for each developer will be 1). The reasons we adopt the probabilities are that, the ones are commonly used in bug assigning methods [35, 2, 18, 21], they can take into account the contents in the descriptions of bug reports, and statistically measure the appropriateness of the task for developers.

To calculate the probabilities, we use a Support Vector Machine (SVM) [46] while there are numerous machine learning algorithms such as Naive Bayes [45], C4.5 [101] and so forth. SVM offers strong performance on unknown patterns (high generalization ability) [46] and it would help RAPTOR assign bug reports including a wide variety of words. A prior study [2] has indicated that SVMs are the most accurate for bug assignments.

SVM searches for a separating hyperplane that separates the positive and negative

examples with the largest margin. The equation of the hyperplane is defined as $\mathbf{w} \cdot \mathbf{x} + b$. The margin is the shortest distance between the positive and negative examples that are the closest to the hyperplane. Searching for the maximum margin is equivalent to finding the minimum norm of \mathbf{w} . The SVM training can be formulated as the following optimization problem

$$\min_{\mathbf{w}} \frac{1}{2} \|\mathbf{w}\|^2, \quad (5.5)$$

$$\text{s.t.} \quad y_i(\mathbf{x}_i \cdot \mathbf{w} + b) - 1 \geq 0, \quad i = 1, 2, \dots, n. \quad (5.6)$$

This problem can be replaced by a dual problem when Lagrangian multipliers are used. By solving the dual problem, we obtain the sets of α_i that minimize the following formulation

$$\max_{\alpha} \quad -\frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j + \sum_{i=1}^n \alpha_i, \quad (5.7)$$

$$\text{s.t.} \quad \sum_{i=1}^n \alpha_i y_i = 0, \quad (5.8)$$

$$0 \leq \alpha_i, \quad i = 1, 2, \dots, n. \quad (5.9)$$

Moreover, the inner product x_i and x_j in the above equation can be substituted by kernel functions. In this study, we adopt a polynomial kernel in the following manner.

$$K(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i \cdot \mathbf{x}_j)^d \quad (5.10)$$

To obtain the output (the probability of appropriate developers), inputs are provided to the SVM in the form of descriptions in the bug reports. The descriptions are preprocessed via tokenization, lemmatization, and stopword removal. Finally, the texts are vectorized by bag-of-words and weighted with TF-IDF. However, SVM training constructs a binary classifier. To decide who should be assigned, we require multi-class classification. Therefore, we conduct pair-wise classification and subsequently calculate the probability of each class (developer) with the pairwise coupling proposed by Hastie and Tibshirani [102]. During the training phase, we employ sequential minimal optimization [103].

Fig. 5.2 shows the preference calculation procedure, and the steps are as follows.

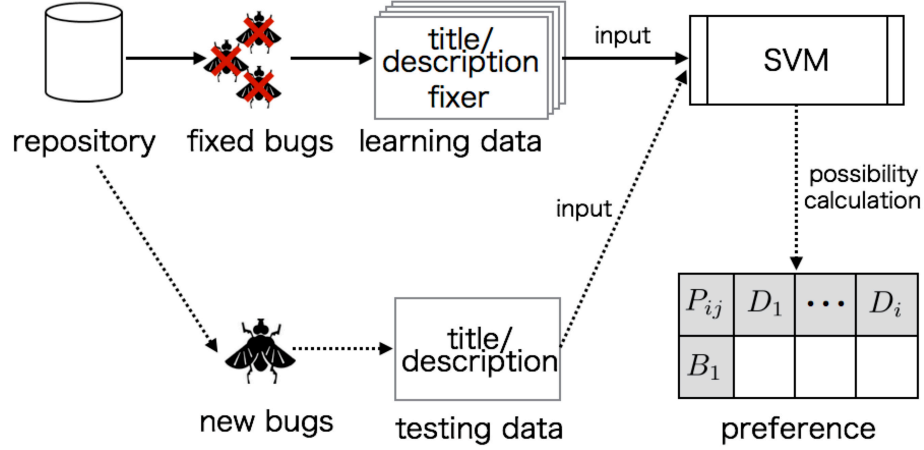


Figure 5.2: The calculation procedure for preference. Preference is calculated by a Support Vector Machine trained with text data and the name of the fixer as a label.

Preparation phase

1. Collect data on fixed bugs from the BTS.
2. Retrieve the fixer and bug description (title and overview) from the data.
3. Train the SVM using fixer/description pairs.

Assignment phase

1. When a new bug B_j is reported, input its description to the previously-generated SVM to obtain the probability (preference P_{ij}) that each developer D_i is suitable for fixing it.

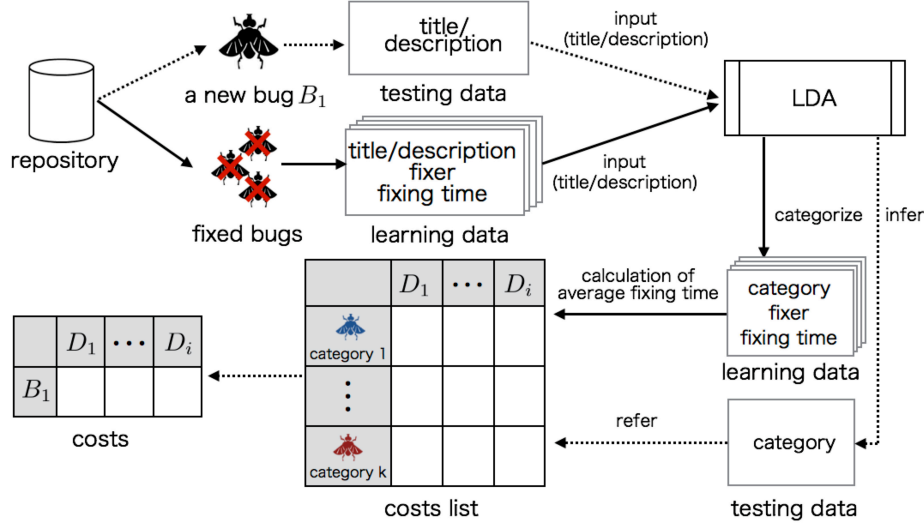


Figure 5.3: The calculation procedure for the bug-fixing costs. The bugs are classified into categories, which indicates the median amount of bug-fixing days and thus the cost of fixing.

Bug-fixing cost

The time required to fix bugs depends on which developers are assigned to fix them. Here, the time required for the developer D_i to fix the bug-fixing task B_j is defined as the **bug-fixing cost** C_{ij} . We use historical data to calculate how long it took for developer D_i to fix similar bugs B_j and use this as the cost C_{ij} . In our previous work [104], in order to calculate the approximate bug fixing-time for the costs, we used priority and component tags. Both tags are located in bug tracking systems, the priority tags show the importance of the bug-fixing and component tags indicate which software parts including the bug. We calculated the median time of bug-fixing as costs by the levels of priority in each component. However, in addition to the calculation of the preference, the calculation of the costs do not use the contents of the bugs, in other words, it ignores what bug it is. Depended on the contents, the bug-fixing time will vary. For example, bugs related to security are fixed faster than bugs about performance [65]. However, component tags do

not include such information and only the description contains the information. Utilizing the descriptions, we use latent Dirichlet allocation (LDA) [105], which is used in [18], to assess whether a previous bug is of a type similar to the current bug B_j . Since LDA is useful for finding similar documents, it is widely used in the mining software repositories field [106]. The latent semantic indexing (LSI) [107] and pLSI [108] methods are similar to LDA, but LDA is different in that the words and topics are assumed to follow a Dirichlet distribution. The fact that it can handle words that were not in the training set is also very useful, and we chose it due to the high probability of new words appearing in the free description part of the input defect forms. Fig. 5.3 shows the bug-fixing cost calculation procedure, and the steps are as follows.

Preparation phase

1. Collect the bug-fixing data from the BTS.
2. Retrieve the fixer and bug description (title and overview) from the data.
3. Input the free description part of the extracted data to the LDA and categorize the bug (as category k).
4. Calculate the average time taken for each developer to fix bugs in each category (called the cost list).

Assignment phase

1. When a new bug B_j is reported, input its description to the previously-generated LDA and infer the bug's category.
2. Find the average time taken by developer D_i to fix bugs in category k from the cost list and use that as the bug-fixing cost C_{ij}

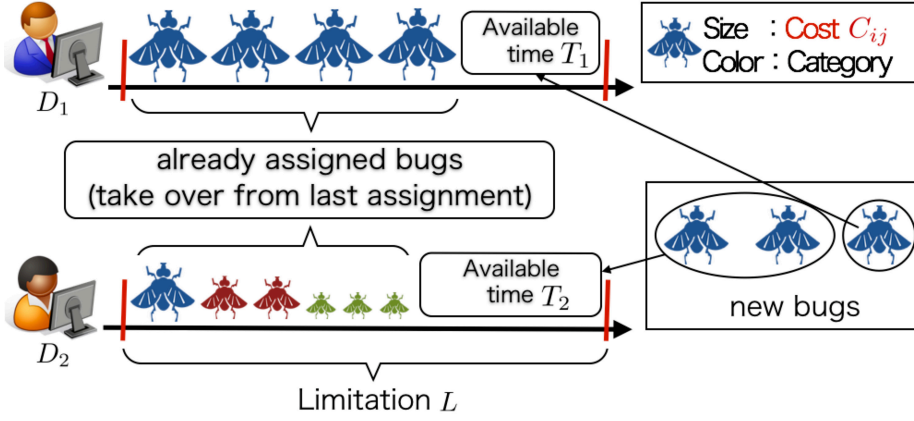


Figure 5.4: Calculation of available time slot. The available time slot T_i is calculated by subtracting the total cost assigned to D_i from the upper limit L (day)

Upper limit

Naturally, the number of tasks developers can address in any given period of time is limited. Thus, when assigning bug-fixing tasks we consider the amount of time that developer D_i has available, i.e., the number of bugs they can fix. Fig. 5.4 shows how the tasks are assigned. The number of tasks that can be assigned is obtained from the **available time slot** T_i , which is calculated from an **upper limit** L (per day) set in advance and the total cost C_{ij} already assigned to developer D_i .

Ensuring that the total cost of the newly-assigned bug-fixing tasks does not exceed T_i should have the effect of preventing these tasks from concentrating on specific developers. The upper limit L can be changed in size depending on the project. We set the same upper limit for all developers in our experiments, but in practice, this upper limit is likely to be different for each developer, in which case it can be set as L_i for developer D_i .

5.3.1 Formulation

Here, we define the objective variables, objective function, and constraints.

Objective variable

The objective variables x_{ij} represent whether our method has assigned bug B_j to developer D_i : if $x_{ij} = 1$, then bug B_j has been assigned to developer D_i , and if $x_{ij} = 0$, then it has not.

$$x_{ij} \in \{0, 1\} \quad (5.11)$$

Objective function

The objective is to maximize the total sum of the product of 10 raised to the power of the level of priority ², the preferences, and objective variables for each bug and each developer. This objective function is designed to find the best combination of tasks and developers for the project as a whole and not for individual developers, and furthermore, bugs with higher priority are prioritized to be assigned.

$$Maximize : \sum_{i=1}^m \sum_{j=1}^n 10^{priority} P_{ij} x_{ij} \quad (5.12)$$

²The highest priority should be assigned the largest value (e.g., P1 (the highest): 5, P5 (the lowest):1)

Constraints

Our method imposes two constraints: one is to prevent tasks from concentrating on a small number of experienced developers (Constraint 1), and the other is to avoid assigning a bug to multiple developers (Constraint 2).

Constraint 1: The total cost of the tasks assigned to each developer must not exceed their available time slots.

$$\sum_{j=1}^n C_{ij}x_{ij} \leq T_i \quad (i = 1, 2, \dots, m) \quad (5.13)$$

Constraint 2: At most, one developer can be assigned to each bug.

$$\sum_{i=1}^m x_{ij} \leq 1 \quad (j = 1, 2, \dots, n) \quad (5.14)$$

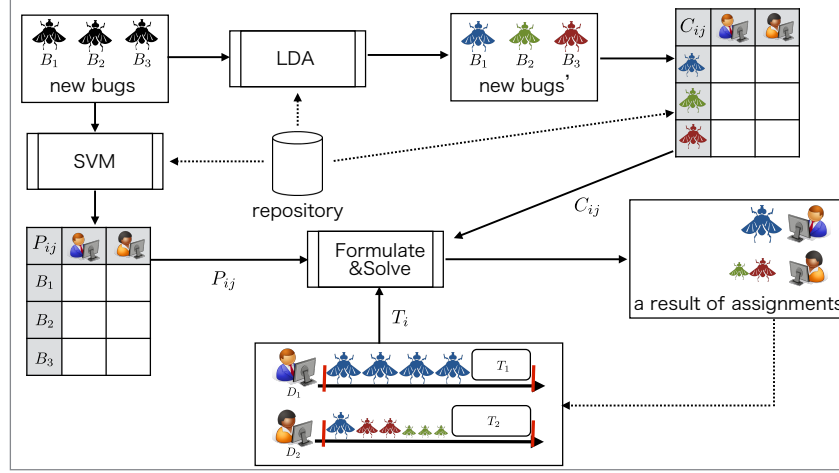


Figure 5.5: Overview of the Release-Aware and Prioritized Task-assignment Optimization Framework (RAPTOR). RAPTOR formulates task-assignment problems with the calculated preference, costs, and available time slots.

5.4 Implementation

5.4.1 Overview of the Proposed Implementation

In this section, we give an overview of the proposed implementation, as shown in Fig. 5.5. First, we extract data about fixed bugs from the repository and use them to train an SVM and an LDA. Next, we obtain the cost list (the average time each developer has taken to fix each category of bugs) using the LDA.

When a new bug B_n is reported, we input its description to the SVM and LDA to obtain the preferences P_{in} for all developers and its category k . Then, we find the costs C_{in} for all developers from the cost list using the category k . Finally, we obtain each

developer's available time slot T_i based on the (pre-determined) upper limit L and the total cost of the bugs already assigned to them. We can then determine the developer to be assigned based on the preferences P_{ij} , the costs C_{ij} and the available time slots T_i .

5.4.2 Procedure for the Release-aware Bug Triaging Method (RAPTOR)

We describe the procedure how to use our method for daily bug assignment as follows.

Step 1: Set the parameters

Set the upper limit L in advance and initialize the available time slots T_i for each developer to L .

Step 2: Construct the SVM and LDA

Construct the SVM and LDA to compute the preferences P_{ij} and costs C_{ij} for each bug B_j and developer D_i .

Step 3: Compute the preferences and costs

Calculate the preferences and costs for each newly reported or unassigned bug.

Step 4: Increment T_i by n (days)

Add n (number of days from the last assignment date to this assignment date) to each developer's T_i (up to a maximum of L). If it is the first assignment, this step will be skipped.

Step 5: Apply 0-1 application of integer programming

Assign these bugs to developers using the method described in the previous section.

Step 6: Update T_i

Reduce the number of available time slots T_i for each developer by the cost of the bugs assigned in Step 4.

Step 7: Go to the next assignment day (to Step 2)

Once the next assignment day comes, proceed to Step 2.

Here, the value of n (> 0) depends on the task assignment process and needs of each

project and is difficult to uniquely determine. In this chapter, we assume that n is a natural number, arbitrarily decided by the method’s users, to keep the discussion general.

5.5 Experimental Design

5.5.1 Overview and Aims

We prepare four evaluations to investigate whether RAPTOR could improve bug-fixing efficiency by assigning tasks to appropriate developers and considering the time they had available for bug-fixing. In Evaluation I, we make sure whether RAPTOR can prevent tasks being concentrated on certain developers, with comparing the existing methods. In Evaluation II, comparing the existing methods, we confirm that RAPTOR can reduce the numbers of overdue bugs (which are assigned but fixed after the release). In Evaluation III, we compare manual assignment (actual bug-fixing time) with the existing methods and RAPTOR to see whether they could reduce bug-fixing delays. In Evaluation IV, we check whether the existing methods or RAPTOR can assign bugs to suitable developers and prevent reassignment, which is the most significant cause of bug-fixing delays. Note that we do not compare our previous work [104] with the proposed work in this study and other existing works although this evaluation might show the difference between our current work and our previous work. This is because our previous method use components tag in the bug reports to assign bugs, which is far from typical bug-triage methods. Most of the bug-triaging studies use the description of the bugs to assign bugs and use the components to measure whether the assignment is appropriate. Basically, developers assign the bugs after reading the description of bug reports rather than component tags, therefore, using the description would be more realistic.

Table 5.2: Datasets

Project	Dataset	Date	Period	No. of bugs
Firefox	Training data	Jul. 5, 2010 – Jul. 4, 2011	1 year	1,043
	Testing data	Jul. 5, 2011 – Sep. 27, 2011	12 weeks	142
Eclipse Platform	Training data	Mar. 22, 2010 – Mar. 21, 2011	1 year	783
	Testing data	Mar. 22, 2011 – Jun. 22, 2011	3 months	194
GCC	Training data	Dec. 25, 2009 – Dec. 24, 2010	1 year	940
	Testing data	Dec. 25, 2010 – Mar. 25, 2011	3 months	250

Table 5.3: Dataset contents after each filtering step

	Filter	Firefox		Eclipse Platform		GCC	
		Training	Testing	Training	Testing	Training	Testing
A	Bugs collected from each project	14,915	2,254	3,503	847	4,191	1,173
B	Of (A), fixed bugs with known fixing-times	1,769	238	1,121	250	1,324	377
C	Of (B), bugs whose fixing-time is not an outlier value	1,568	211	946	202	1,116	320
D	Of (C), bugs fixed by active developers	1,043	142	783	168	940	250

Table 5.4: Statistics of fixing-time for each dataset

Project	Firefox	Platform	Gcc
# of bugs	1,185	951	1,190
Percentage of bugs in which the bug-fixing time is less than 2 days	19.2	49.2	51.2
average days to fix	12.5	6	4.4
median days to fix	7	2	1.9
minimum days to fix	1	1	1
maximal days to fix	59.9	38.5	27.5

5.5.2 Datasets

We conducted a case study on three large OSS projects (Mozilla Firefox ³, the Eclipse Platform⁴, and GNU compiler collection (GCC) ⁵). Each of these is a long-established project, allowing us to acquire sufficient data for the experiment. In addition, many previous studies [2, 11, 13, 18, 21, 35, 37, 109, 110, 111] have analyzed data from these projects, enabling us to validate the results obtained in this case study.

Table 5.2 outlines the datasets used, whereas Table 5.3 lists the filtering performed to create them and the number of bugs in each. Table 5.4 shows their statistics of bug-fixing days. Of all the bugs collected from each project (Filter A in Table 5.3), we only considered fixed bugs (i.e., bugs whose status was FIXED) where the fixer and fixing time could be identified (Filter B). Some of the bugs were only fixed after several years, so we removed these outliers by confirming the fixing time distributions using boxplots (Filter C).

In this study, we assigned bug-fixing tasks to developers using existing methods and RAPTOR. However, assigning tasks to all of a project’s developers is not realistic because OSS projects developers are often known to leave projects in a relatively short period

³Mozilla Firefox: <https://www.mozilla.org/en-US/firefox/new/>, Last Accessed: January 2020

⁴Eclipse Platform: <https://projects.eclipse.org/projects/eclipse.platform>, Last Accessed: January 2020

⁵GNU Gcc: <https://gcc.gnu.org/>, Last Accessed: January 2020

of time [112]. Moreover, since not all developers actively fix bugs [109], tasks should necessarily only be assigned to developers who are likely to be in charge of bug-fixing tasks. Hence, we only assigned defined tasks to developers who had fixed six or more bugs within six months of their first assignment (i.e., fixed at least one bug per month), thus considering these developers to be “active” (Table 5.5). To guarantee the accuracy of the task assignment, all bug reports fixed by non-target developers were excluded (Filter D).

In this study, we prepared both learning and evaluation datasets, using one year of data (from the first assignment day) as training data for all projects, and 12 weeks of data (Firefox) and three months of data (Eclipse and GCC) before release as evaluation data. Only 12 weeks of Firefox evaluation data was used because the Firefox project has adopted a rapid release method [113] with a test period of 12 weeks for each release. In contrast, three months of evaluation data of Eclipse and GCC was used due to a large number of bug reports filed in the three months before release.

5.5.3 Comparison Methods

We compared the bug-fixing time of RAPTOR with that of a manual assignment method and two existing methods (CBR and CosTriage). Among the machine learning algorithms used for CBR and CosTriage, we used the SVM-based method [2] that was found to give the most accurate recommendations.

5.5.4 Evaluations

We evaluate RAPTOR in four different ways as described below.

Evaluation I: Prevention of task concentration

We confirm whether the number of tasks (bug-fixing time) assigned to each developer by the existing methods and RAPTOR is higher for certain developers. Here, as an evaluation criterion, the bug-fixing time should not exceed the evaluation data period for each project.

Table 5.5: Active developers in each dataset

Projects	# of all developers	# of active developers
Firefox	215	19
Platform	61	20
Gcc	97	23

Evaluation II: Reducing overdue bugs for the release

We confirm the numbers of bugs that assigned but fixed after the release (“# of overdue bugs”). Note that “# of overdue bugs” is different from the task concentration in Evaluation I, which shows the total time that developers devote fixing bugs in the period. Therefore, even if the task concentration did not happen in Evaluation I, if the bugs were assigned immediately before the release, “# of overdue bugs” might be more than zero.

Evaluation III: Reduction of overall bug-fixing time for the project

We confirm whether the existing methods or RAPTOR can improve bug-fixing efficiency by comparing their estimated bug-fixing times with the actual recorded times.

Evaluation IV: Accuracy of assignments

We evaluate to what extent the accuracy of assignments by RAPTOR decreases compared to CBR. CBR assigns each bug to the most suitable developer (with the largest preference), whereas RAPTOR assigns bugs to developers so that the total preferences for the project are the highest. Hence, we can assume that RAPTOR will lower the accuracy of the assignment.

The accuracy of assignments measures a rate of the number of appropriate assignments and the number of all assignments. The appropriate assignment is defined as an assignment to the developer who has experienced fixing bugs with the same component as the target bug report. The components are software parts constituting the product. The bug tracking systems in Eclipse, Firefox, and Gcc has the tag indicating which component includes the bug.

Here, several works often evaluate their methods with top-X accuracy which is the performance measure how many developers are selected when recommending multiple

developers for a bug. We cannot use this because our bug assignments to developers are executed at the same time and determined dynamically, that is, each assignment is dependent on to whom each other's bug is assigned.

5.5.5 Experimental Procedure

In this experiment, tasks were assigned using both the existing methods and RAPTOR, and the bug-fixing times were calculated based on the resulting assignments. An overview of the experiment is shown in Fig. 5.6.

We extracted the bug reports for each date in the evaluation data and used both RAPTOR and the existing methods to assign the bugs day by day according to their reported date. Also, the assigned bugs to each developer are considered to be fixed in the order of the assignments. Once the methods finish assigning bugs each day, developers' available time slots T_i will be incremented by one (T_i never surpass upper limit L).

Once assignments had been made for all days, the bug-fixing times were calculated (Fig. 5.6, right). Since the assignment methods considered here do not always assign the bugs to the developers who actually fixed them (i.e., the actual bug-fixing time cannot be calculated), we used the median times taken by the individual developers to fix the bugs in each category (that is, the costs C_{ij}) from the training data as the bug-fixing times for the experiment.

Experimental environment and settings

Experimental environment: The open-source mathematical planning software package lp_solve 5.5.2.0 is used to solve the task assignment problem using 0-1 integer programming method, operating on a PC with an Intel Xeon 2.20 GHz CPU and 64 GB of RAM and running CentOS 7.

Parameter settings: RAPTOR requires the upper limit L and the assignment interval (Section 5.4.2, Step 6, n) to be set in advance. Here, the third quartile value of the times required to fix the bugs in the dataset was calculated and rounded, to obtain L values of 15 for Firefox, 6 for the Eclipse Platform, and 6 for GCC. In addition, the

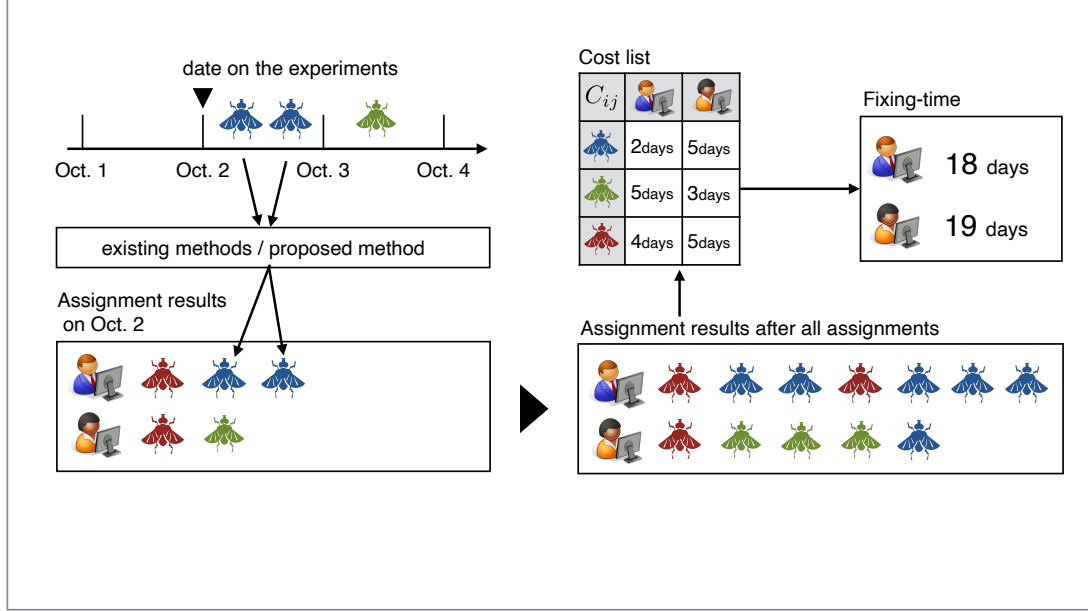


Figure 5.6: Overview of our experiment. The number of fixing days are calculated by multiplying the cost with the assignment result.

interval n was set to 1 (day). While applying LDA, deciding how many bug categories to use for classification was important, so we determined the optimal number of categories for each project using Arun’s method [114]. This yielded 7 for Firefox, 12 for the Eclipse Platform, and 11 for GCC.

Experimental settings: The procedure of RAPT includes the recalculation process of preferences and costs (Step2). However, if we use this recalculation in the evaluation, we cannot compare the three methods under the same condition because the preference and cost gradually vary as the simulation progresses. In order to prevent from changing the cost and preference during the experiment, we return to Step 3 rather than Step 2 after Step 6 in this experiment.

As in a previous study [18], the bug-fixing times for prior bugs were obtained as follows.

$$\text{fixing time} = \text{fix day} - \text{assignment day} + 1 \text{ day} \quad (5.15)$$

*round down to the nearest decimal

The assignment date is the date when the bug was assigned to the developer who fixed it. In other words, we do not include the time spent by previous developers in attempting to fix the bug (the reassignment time) here.

5.6 Results

5.6.1 Preliminary Experiment: Evaluation of a Method to Calculate Bug-fixing Time

To the best of our knowledge, no software provides us with a simulation of bug-fixing activity. In this experiment, we can not time the bug-fixing time if tasks are assigned to developers other than the developer who actually fixed the bug. This is the reason why costs are substituted as bug-fixing time in this experiment. To use the cost as the bug-fixing time, we confirm whether the cost can substitute as the bug-fixing time in a preliminary experiment.

First, from the bug-fixing history, we prepare two kinds of information (“who fixed which bugs” and “the fixing time”). Using the former information (who fixed which bugs) and cost, we calculate the simulation bug-fixing time. Then, we compare the simulation bug-fixing time and the actual bug-fixing time. As the difference between these two fixing times is smaller, the calculation of the bug-fixing time in this simulation is reasonable.

The results of the experiment are shown in the Table 5.6. In Firefox 156 days (1.1 days per bug), Platform 44 days (0.2 days), and Gcc 156 days (0.6 days) errors were seen. The error per bug is about one day in Firefox or is less than one day in Platform and Gcc, and can be said to be acceptable.

Table 5.6: Evaluation of calculations for bug-fixing time

	Actual fixing day	Simulated fixing day	delta	delta per bug
Firefox	1,799	1,643	156	1.1
Platform	822	778	44	0.2
Gcc	1,148	992	156	0.6

5.6.2 Evaluation I: Mitigation of Task Concentration

Fig. 5.7 shows the amount of task (the number of days to fix) that each active developer worked on ⁶. The number of developers assigned tasks that require more than the evaluation data period (Firefox: 12 weeks, Platform and Gcc: 3 months) by CBR, is eight developers in Firefox, one developer in Platform, four developers in Gcc. We can see a lot of loads on some developers.

Even when using CosTriage, since the number of developers is seven developers in Firefox, no developers in Platform, two developers in Gcc. It shows Costriage mitigate the task concentration compared with CBR, however, tasks are still concentrated on a few developers. In the case of applying RAPTOR, the number of developers is two developers in Firefox, no developers in Platform and Gcc. For all projects, the number of developers is reduced compared with existing methods.

As for the bug-fixing times of the developers who concentrated tasks by RAPTOR and the existing methods, it can be seen that the bug-fixing time of the developer assigned by RAPTOR is significantly reduced (especially, the fixing-times of developers assigned a lot of tasks by existing methods are reduced).

Table 5.7 summarizes the statistics of the fixing-time that each developer devotes to fixing bugs. The variance of the fixing-time assigned by RAPTOR is smaller than the others in all projects and also entropy is larger, which show RAPTOR can mitigate the

⁶Note that the numbers on the horizontal axis represent the order when the developer's task amount (bug-fixing days) is arranged in descending order for each method, therefore different developers even if same axis numbers for each method.

tasks more than the traditional methods do.

The existing methods tend to concentrate the task assignment on some developers. Compared with the existing methods, RAPTOR can mitigate the task concentration.

5.6.3 Evaluation II: Reducing Overdue Bugs for the Release

Table 5.8 shows the numbers of bugs that assigned but fixed after the release (“# of overdue bugs”) and the numbers of unfixed bugs by the release which is the sum of “# of overdue bugs” and “# of un-assigned bugs”.

In Firefox, 9 by the manual assignment method, 77 bugs by CBR, 75 bugs by CosTriage, and 31 bugs by RAPTOR are overdue. Next, in Platform, 5 by the manual assignment method, 20 bugs by CBR, 19 bugs by CosTriage, 5 bugs by RAPTOR. Finally, in Gcc, 22 by the manual assignment method, 48 bugs by CBRs, 38 bugs by CosTriages and 15 bugs by RAPTOR. Overall, RAPTOR can assign a more appropriate amount of bugs to each developer compared to the existing methods.

We looked into the assigned date of the overdue bugs and found that the overdue bugs by RAPTOR were reported and assigned just before release. For the existing methods, in addition to the reason, the task concentration made overdue bugs.

Compared with existing methods, the number of unfixed bugs by RAPTOR is fewer than CBR and Costriage in Firefox and Gcc but is more than CBR and Costriage in Platform. The number by the manual assignment is the largest in Platform and Gcc while the number is the least in Firefox among the four methods.

Moreover, we show the details of unassigned, overdue, and unfixed bugs by the level of priority in Table 5.9. Looking at the higher priority (P1 and P2) of unfixed bugs, the total number by RAPTOR is the smallest in Firefox and Gcc while one bug of P2 remains in Platform only by RAPTOR. We discuss how to handle these unfixed bugs with high priority in the discussion session.

RAPTOR can assign a more appropriate amount of bugs that each developer can fix by the immediate release, compared with CBR and CosTriage.

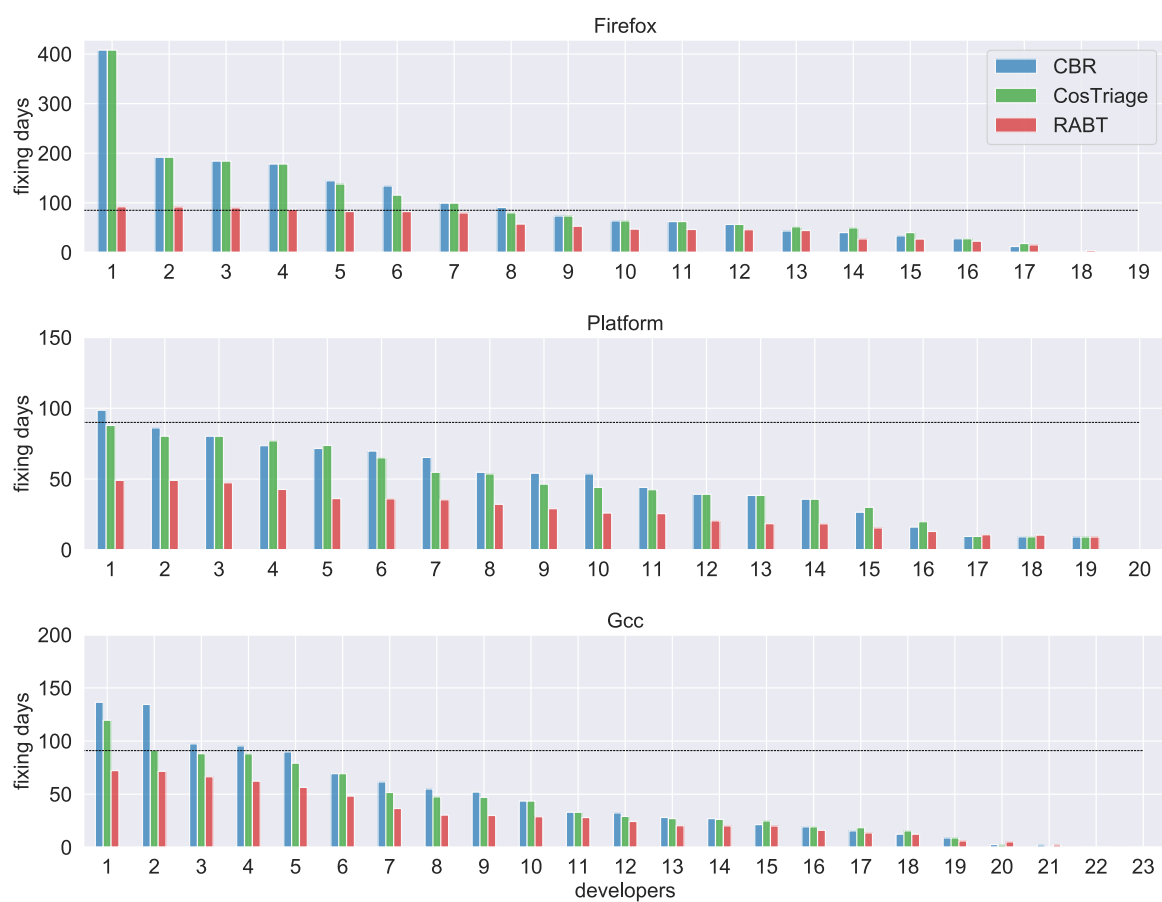


Figure 5.7: Fixing days by each developer.

Table 5.7: The statistics of fixing-time assigned by each method

Projects	Firefox		
Methods	CBR	CosTriage	RAPTOR
mean	97.0	96.7	52.3
median	63.6	63.6	46.9
max	407.9	407.9	92.1
min	0.0	0.0	0.0
variance	9393.3	9117.8	952.1
stdev	96.9	95.5	30.9
entropy	3.6	3.7	4.0

Projects	Platform		
Methods	CBR	CosTriage	RAPTOR
mean	46.8	44.8	26.2
median	48.9	43.3	25.8
max	98.6	87.9	49.1
min	0.0	0.0	0.0
variance	828.3	716.7	210.6
stdev	28.8	26.8	14.5
entropy	4.0	4.0	4.1

Projects	Gcc		
Methods	CBR	CosTriage	RAPTOR
mean	45.1	40.4	29.1
median	32.4	29.1	24.3
max	136.5	119.6	72.2
min	0.0	0.0	0.0
variance	1716.4	1167.7	536.4
stdev	41.4	34.2	23.2
entropy	3.9	4.0	4.0

5.6.4 Evaluation III: Reduction of Bug-fixing Time

Table 5.8 shows the bug-fixing time of the project when using the manual assignment method, CBR, CosTriage, and RAPTOR. Fixing-days for projects do not include the time of un-assigned bugs⁷, therefore, we calculate the average fixing-days per bug.

In Firefox, the average bug-fixing days per bug is 11.8 days in the manual assignment method, 13.0 days in CBR, 12.9 days in Costriage, 8.1 days in RAPTOR. CBR increased about 10% ($10\% = (13.0 - 11.8) / 11.8$) of the days compared with the manual assignment method, Costriage also raised about 9% ($9\% = (12.9 - 11.8) / 11.8$), RAPTOR could reduce about 31% ($-31\% = (8.1 - 11.8) / 11.8$) compared to the manual assignment method. Moreover, RAPTOR could reduce about 38% ($-38\% = (8.1 - 13.0) / 13.0$) of the days compared to CBR, and about 37% ($-37\% = (8.1 - 12.9) / 12.9$) compared to CosTriage.

In Platform, the total number of bug-fixing days for the project is 4.6 days in the manual assignment method, 4.8 days in CBR, 4.6 days in Costriage, 3.0 days in RAPTOR. CBR increased about 4% ($4\% = (4.8 - 4.6) / 4.6$) of the days compared with the manual assignment method, Costriage does not reduce ($0\% = 4.6 - 4.6 / 4.6$), RAPTOR could reduce the bug-fixing time of about 35% ($-35\% = (3.0 - 4.6) / 4.6$) compared with the manual assignment method. In addition, RAPTOR could reduce the bug-fixing time of about 38% ($-38\% = (3.0 - 4.8) / 4.8$) compared to CBR, while RAPTOR increased about 35% ($-35\% = (3.0 - 4.6) / 4.6$) of the days compared to CosTriage.

In Gcc, the total number of bug-fixing days for the project is 4.6 days in the manual assignment method, 4.1 days in CBR, 3.7 days in Costriage, and 2.8 days in RAPTOR. CBR could reduce about 11% ($-11\% = (4.1 - 4.6) / 4.6$), about 20% ($-20\% = (3.7 - 4.6) / 4.6$), RAPTOR can reduce the bug-fixing time of about 39% ($-39\% = (2.8 - 4.6) / 4.6$) compared to the manual assignment method. In addition, RAPTOR could reduce the bug-fixing time of about 32% ($-32\% = (2.8 - 4.1) / 4.1$) compared with CBR, while increased about 24% ($-24\% = (2.8 - 3.7) / 3.7$) compared to CosTriage.

Compared to the manual task assignment method, RAPTOR can reduce the bug-fixing time from 31% to 39%.

⁷Because of this, the fixing days of the manual method in Table 5.8 is different from that in Table 5.6

Table 5.8: Comparing the results of each method

Projects	Firefox			
Methods	Manual	CBR	CosTriage	RAPTOR
# of assigned bugs	113	142	142	123
# of un-assigned bugs	29	0	0	19
# of assigned developers	15	17	17	18
# of overdue bugs	9	77	75	31
# of un-fixed bugs	38	77	75	50
Fixing-days for project	1,338	1,843	1,838	994
Avg. Fixing-days per bug	11.8	13.0	12.9	8.1
Accuracy of assignments	—	81.7	81.0	70.7
Projects	Platform			
Methods	Manual	CBR	CosTriage	RAPTOR
# of assigned bugs	146	194	194	177
# of un-assigned bugs	48	0	0	17
# of assigned developers	19	19	19	19
# of overdue bugs	5	20	19	5
# of un-fixed bugs	53	20	19	22
Fixing-days for project	669	936	897	525
Avg. Fixing-days per bug	4.6	4.8	4.6	3.0
Accuracy of assignments	—	68.0	68.0	62.7
Projects	Gcc			
Methods	Manual	CBR	CosTriage	RAPTOR
# of assigned bugs	206	250	250	243
# of un-assigned bugs	44	0	0	7
# of assigned developers	19	21	20	21
# of overdue bugs	22	48	38	15
# of un-fixed bugs	66	52	40	22
Fixing-days for project	940	1,037	929	670
Avg. Fixing-days per bug	4.6	4.1	3.7	2.8
Accuracy of assignments	—	74.4	71.6	67.9

Table 5.9: The details of unfixed bugs

	Project Methods	Firefox			
		Manual	CBR	CosTriage	RAPTOR
# of unassigned bugs	P1	3	0	0	3
	P2	7	0	0	1
	P3	19	0	0	15
	P4	0	0	0	0
	P5	0	0	0	0
# of overdue bugs	P1	2	4	4	1
	P2	0	6	6	2
	P3	7	65	63	17
	P4	0	0	0	0
	P5	0	0	0	0
# of unfixed bugs	P1	5	4	4	4
	P2	7	6	6	3
	P3	26	65	63	32
	P4	0	0	0	0
	P5	0	0	0	0
	Project Methods	Platform			
		Manual	CBR	CosTriage	RAPTOR
# of unassigned bugs	P1	0	0	0	0
	P2	0	0	0	1
	P3	48	0	0	16
	P4	0	0	0	0
	P5	0	0	0	0
# of overdue bugs	P1	0	0	0	0
	P2	0	0	0	0
	P3	5	20	19	5
	P4	0	0	0	0
	P5	0	0	0	0
# of unfixed bugs	P1	0	0	0	0
	P2	0	0	0	1
	P3	53	20	19	21
	P4	0	0	0	0
	P5	0	0	0	0
	Project Methods	Gcc			
		Manual	CBR	CosTriage	RAPTOR
# of unassigned bugs	P1	0	0	0	1
	P2	3	0	0	0
	P3	40	0	0	6
	P4	1	0	0	0
	P5	0	0	0	0
# of overdue bugs	P1	0	0	0	0
	P2	4	4	4	1
	P3	18	41	31	13
	P4	0	3	3	1
	P5	0	0	0	0
# of unfixed bugs	P1	0	0	0	1
	P2	7	4	4	1
	P3	58	41	31	19
	P4	1	3	3	1
	P5	0	0	0	0

5.6.5 Evaluation IV: Accuracy of Assignments

Table 5.8 also shows the accuracy of assignments by CBR, CosTriage and RAPTOR, respectively. In Firefox, the accuracy of assignments was 81.7% by CBR, 81.0% by CosTriage, 70.7% by RAPTOR. Next, in Platform, CBR was 68.0%, CosTriage was 68.0%, and RAPTOR was 62.7%. Finally, in Gcc, CBR was 74.4%, CosTriage was 71.6%, and RAPTOR was 67.9%. Taking the average accuracy for each of the three methods, CBR is 74.7%, the CosTriage is 73.5%, RAPTOR is 67.1%. In addition, the accuracy of CosTriage was 2% lower than that of CBR, and the accuracy of RAPTOR decreased by 11% compared to CBR.

Although RAPTOR can reduce the bug-fixing time and mitigate the concentration of tasks, it has been found that the assignment accuracy decreases by 11% on average, compared to CBR.

5.7 Discussions

5.7.1 The Cause of Lowering Accuracy

Throughout the evaluations, we showed RAPTOR outperforms the existing methods in terms of mitigation of the task concentration, the number of bugs that developers can fix by the next release, the total fixing-time for the project. However, in evaluation IV, we found RAPTOR decreases 11% of the accuracy of assignments, comparing with the existing methods. We concern the effect of lowering the accuracy which induces the reassignments of the bugs. We have two conceivable reasons why the accuracy of RAPTOR is lower. The first case is when we still have the other developers that have fixed a bug in the same component (which would be a correct recommendation if methods assign a bug to the developers). RAPTOR would assign bugs to the developers whose preference is not the largest. Thus, in the case that the assignments are inaccurate even though there are alternative developers, this suggests that the second or the third (or so on) recommendations should be improved. Since the number of fixes is considerably different depending on developers, the sizes of the training dataset for each developer also differ. In this experiment, we used the dataset which contains one year of data. This is because the dataset size becomes bigger, the more the existing methods would concentrate bugs on specific developers, therefore we avoid using plenty of the data to equally evaluate. If we use more data, the second or third recommendations would be improved.

Another case is when there is only one developer (there are no alternative developers) in the component. In this case, the preference of the developers that should be assigned by RAPTOR should be 100 (which is the maximum value of preference). To realize the value, we could train the model with the component tags in addition to text data. Although the component tags were used to evaluate the appropriateness of the recommendations in this chapter, we can exploit the tags when applying RAPTOR to the actual projects. For both cases, improving the classifier would be an effective option.

5.7.2 How to Handle Unfixed Bugs with High Priority?

Evaluation II shows that RAPTOR decreases the number of unfixed bugs in all projects, compared with existing methods. However, even assigning with RAPTOR, the unfixed bugs with high priority remain in all projects. We investigated the reason why the bugs remain, looking into the preference and cost. We found that these bugs have higher costs for all developers than the maximum Limit L , which did not allow RAPTOR to assign.

In this case, we are able to easily detect the bugs with higher costs than L by monitoring the costs before assigning them. Taking into consideration of applying to practical projects, we should implement a function to prompt managers to assign the bugs manually.

5.7.3 How to Set Appropriate Limit L ?

In this experiment, we set the upper limit $L = 6$ for Platform and Gcc, 15 for Firefox. However, it is unclear what impact the size of the upper limit L has on the project. Hence, we confirm how the accuracy of assignments and the number of overdue bugs vary with the size of the upper limit L . Fig. 5.8 is the accuracy of assignments and the number of overdue bugs for each upper limit L . The accuracies dynamically increase from 1 to 11 for Firefox, from 1 to 4 in GCC. After that, the accuracies gradually increase until L is 31. In Platform, the accuracy does not dynamically change across L . In terms of the number of unfixed bugs, in Firefox and Gcc, the number dynamically drop between 1 and 15 in Firefox, 1 and 4 in Platform and Gcc.

We can see the best points of L should be of which the accuracy and the number of unfixed bugs calm down. The points are between 15 and 19 in Firefox, 7 and 23 in Platform, 5 and 7 in GCC. Our way to set L , which is referring to the 3rd quartile (Firefox:15, Platform:6, Gcc:6), might be appropriate, considering the cost of deciding L .

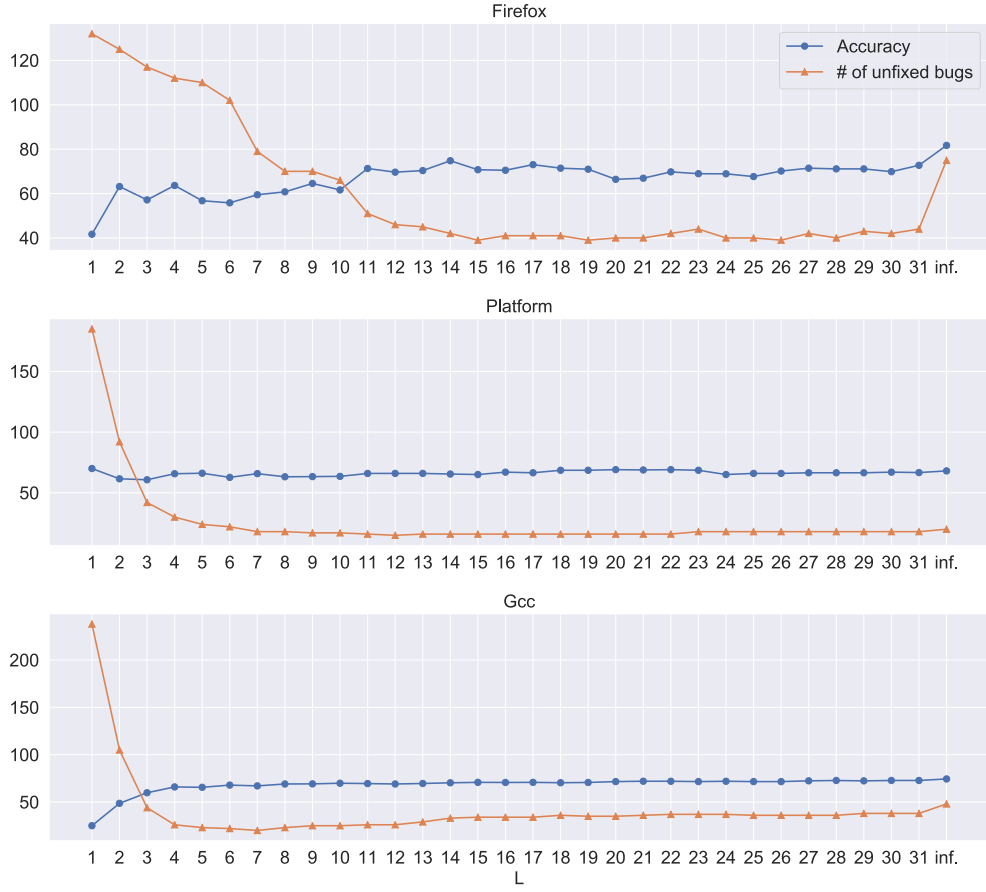


Figure 5.8: Relationship between the size of L and the accuracy and the number of overdue bugs

5.7.4 How to Deal With Irregular Situations

As RAPTOR is designed for automating usual bug assignments, developers would have to help RAPTOR to assign bugs if unexpected problems happened. In the following, we discuss about likely irregular situations.

Nobody can fix the bug in the project: This situation is likely to happen to any other bug-assignment methods including manual bug-triage, but practically RAPTOR should be more careful about this situation because it aims to automate bug assignments. In the RAPTOR, the preference is relative scores, produced by Support Vector Machine. That is the total of the possibility for each developer will be 1. Thus, even if there is nobody who is appropriate, RAPTOR (but also CBR and CosTriage) can choose developers who are relatively appropriate among all developers. Nevertheless, it is probable that the developer cannot fix the bug. In the case that the assigned developer cannot fix the bug, in advance, the project has to make the rule that the assigned developer should lead the other members and discuss how to handle the bug. In accordance with the situation, the project would need to call for other professionals from outside of the project.

Developers are faced with technical or private problems: In the procedure of RAPTOR, on every assignment, available time slots (T_i) is incremented by the days from the last assignment day to the assignment day. However, if unexpected problems happened, there is a probability that such simple incrementing might not correctly reflect their workloads. For example, if a developer is taking more time than the estimated time because of technical or private problems, RAPTOR will continue to assign new bugs. In case of unexpected problems, RAPTOR needs a function to stop assigning them (and/or reassign the bugs to the others) when RAPTOR is implemented for applying to practical projects. As a better option instead of the simply increments function (Step 4) every assignment for the available time slot, RAPTOR could replace a new update function that removes the occupied cost after the bug is fixed in the available slot. However, even with this step, since the case developers cannot fix the bug will happen, the reassignment function should be required.

5.7.5 Limitation

Fix Order of Bugs

In Experiment II, by comparing the number of bugs fixed by the release with RAPTOR and the two existing methods, we confirmed that the existing methods remain many bugs not fixed by the release. However, the number of bugs not fixed by the release depends on the order in which the bugs are fixed. In other words, if the bugs which are long bug-fixing time was assigned in the early time of the experiment, the number of the bugs not fixed by the release will increase.

Impact of Mitigating Task Concentration

In the experiment, we could confirm the effect of mitigating the task concentration in RAPTOR. However, mitigating the task concentration of some developers is also that other developers are assigned the tasks. Even though developers who are relatively not in charge of tasks seem to have a scope at first glance, they may have other development projects or volunteers, so the time of activities may be limited. Hence, developers who do not have many tasks are not necessarily in a condition that can handle tasks. Since RAPTOR has not ascertained how long it can participate in the bug-fixing activity in the month, the developer might be forced an excessive load.

5.8 Chapter Summary

In this chapter, we proposed a release-aware bug triaging method (RAPTOR) that aims to increase the number of bugs that developers can fix by the next release date. Existing methods tend to concentrate assignments of bug-fixing tasks to a small number of developers because it does not consider the difficulty and costs of individual bug-fixing. Since general software development has the releases, even an experienced developer can finish the bug-fixing work that can be used until the next release. Hence, the existing methods are not realistic.

RAPTOR is characterized by considering the upper limit of the number of tasks that developers can work on during a certain period, in addition to the ability of developers. In this method, we considered the bug assignment problem as a multi-knapsack problem, finding a combination of bugs and developers that maximize developers' ability under constraints which the method can assign in the only time that developers can use for bug-fixing work. As a result of a case study on Mozilla Firefox, Eclipse Platform, GNU Gcc project, the following three effects on the proposed method were confirmed.

- (1) RAPTOR mitigates the situation where bug-fixing tasks are concentrated to a small number of developers
- (2) RAPTOR can assign a more appropriate amount of bugs that each developer can fix by the next release date
- (3) RAPTOR can reduce time developers devote to fixing bugs, compared with the manual bug triaging method and the existing methods

Chapter 6

Discussions

In Chapter 5, Evaluation II showed that RAPTOR assigns bugs so that it increases the number of fixed high priority bugs. In the experiment, RAPTOR used the actual priority to assign bugs because RAPTOR should be independently evaluated to measure the extent to which the task concentration problem can be mitigated. However, the performance when RAPTOR is combined with cycle-aware priority prediction is yet to be explored. Therefore, in this chapter, we evaluate whether RAPTOR with priority prediction helps projects fix more bugs with higher impact by the release date. We first predict the priorities of bugs with cycle-aware models and then assign bugs using RAPTOR.

Figure 6.1 shows an overview of the datasets used for this evaluation. We reuse the training dataset and testing dataset from Chapter 5 to build RAPTOR and measure its performance, respectively. Subsequently, we train the cycle-aware prediction models with the datasets from Chapter 4. Note that we remove the bugs, which are in the testing dataset, from the training dataset of Chapter 4 because these bugs will be targets to predict and assign. Among various types of cycle-aware models, we build hybrid monthly-cycle-aware models that outperform others.

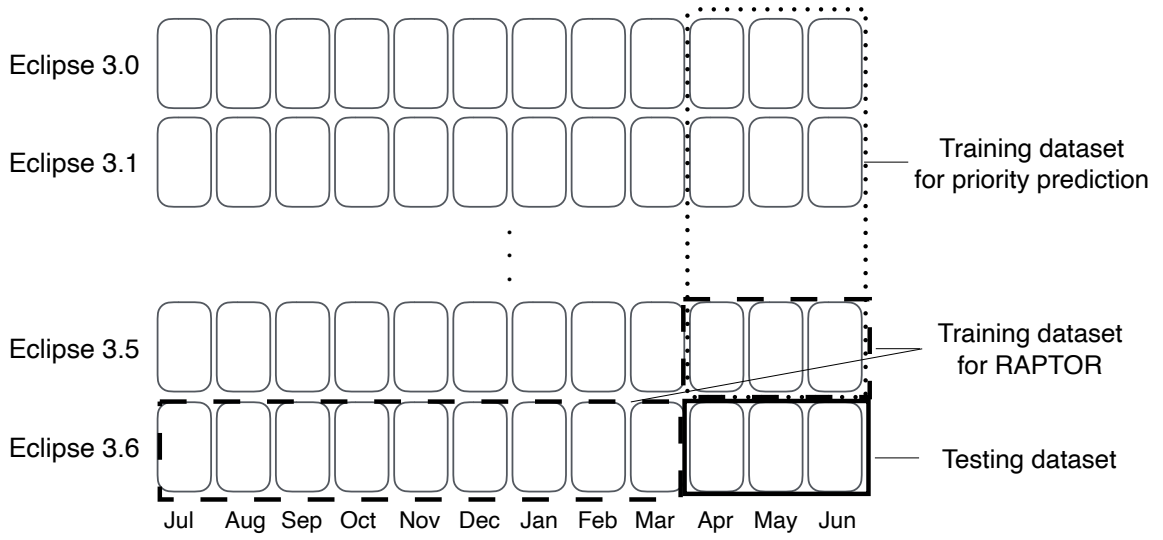


Figure 6.1: Datasets for RAPTOR and the cycle-aware prediction models. We use the data of three months in Eclipse 3.6 for the testing dataset. We build RAPTOR with the data for 1 year previous to the testing dataset and train the cycle-aware models with the data from Eclipse 3.0 to Eclipse 3.5.

Table 6.1: The number of unfixed bugs that the Eclipse project cannot fix by the release date is shown. The table shows the results in the case that RAPTOR assign bugs with two priority prediction models (Opaque model or Cycle-aware models) or without priority predictions. The “Predicted priority” and “Actual priority” columns show the number of bugs with predicted and actual priority, respectively.

	Prediction model	Opaque model		Cycle-aware models		Chapter 5
	Priority	Predicted	Actual	Predicted	Actual	Actual
Assigned bugs	High priority	3	0	12	1	2
	Not high priority	189	186 (3)	180	178 (2)	175
Unfixed bugs	High priority	0	0	3	0	0
	Not high priority	12	12	10	13	22
Unassigned bugs	High priority	0	0	0	0	1
	Not high priority	2	2	2	2	16
Overdue bugs	High priority	0	0	3	0	0
	Not high priority	10	10	8	11	5

Table 6.1 shows the number of bugs unfixed by the release date, which is assigned by RAPTOR with the different priority prediction models, opaque model, cycle-aware prediction models, and without priority prediction (i.e., the result of Chapter 5). The “Predicted” and “Actual” columns show the number of bugs with the priorities predicted by models and the original priorities, respectively.

RAPTOR with the opaque model predicted 3 bugs as high priority bugs and 189 bugs as the others. However, none of 3 bugs did have actually high priority and 3 of 189 had actually high priority. For RAPTOR with the cycle-aware models, 12 and 180 bugs were predicted as high priority bugs and the others, respectively. As a result, 1 bug was truly high priority bug and 2 bugs were missed even though they had high priority tags. RAPTOR with cycle-aware models could detect 1 more bug than that with the opaque model. Compared with the result of Chapter 5, the number of unfixed bugs was reduced but this might be because of the order of bug assignments.

While all the high priority bugs were finally fixed with both priority prediction models,

they wrongly predicted 2 or 3 bugs as normal bugs instead of high priority bugs. In order to fix high priority bugs certainly, the priority prediction models should be improved by proposing new effective metrics to predict. Specifically, to utilize RAPTOR, priority prediction should return positive labels sensitively to prevent a situation in which high priority bugs are missed.

Chapter 7

Conclusion

This thesis constructed the Release-Aware and Prioritized Task-assignment Optimization framework (RAPTOR) in order to increase the number of fixed bugs by the release date. RAPTOR imposes restrictions on assigning the number of bugs to each developer. The limitation would force us to select bugs that will be fixed by the next release. Thus, RAPTOR needs to select which bugs should be fixed by the next release and which bugs should be allowed to carry over to the later release, under the circumstance that numerous bugs are reported. When new bugs are reported, RAPTOR first prioritizes the bugs and then assigns them so that the number of bug-fix with higher priority is increased for the project. The main findings of this thesis are reiterated as follows:

Chapter 3: We asked what kind of bugs developers think impactful and encounter in practice, through a survey with 322 notable GitHub developers. We manually inspected and classified actual bug reports included in the responses. As a result, we showed there is a wide variety of high impact bugs. Particularly, developers think security and breakage bugs are highly important for FLOSS developers. Furthermore, we showed that 66% of the high impact bugs have a higher importance in the projects (especially in the projects that strictly handle bugs). That suggests helping us select bugs for the next release when the projects have a myriad of bugs.

Chapter 4: We predicted the priority which represents the importance of bug-fixing, used by developers. We first examine whether the works (the characteristics) in each period are different and showed that developers' activity varies during the release cycle. Based on these findings, we built release cycle-aware models which are the models of which data is derived from appropriate periods. We conducted a case study on the Eclipse Platform project and found that cycle-aware models outperform the traditional model which uses whole data during the development.

Chapter 5: We constructed RAPTOR and conducted a case study on Eclipse Platform, GNU compiler collection, and Mozilla Firefox and showed that RAPTOR (1) mitigates the situation where bug-fixing tasks are concentrated to a small number of developers; (2) increases the number of high priority bugs by the next release date (3) can reduce the time to fix bugs, compared with the manual bug triaging method and the existing methods.

Publications

Journals

- Yutaro Kashiwa and Masao Ohira, “Release-Aware and Prioritized Bug-fixing Task assignment Optimization,” Transaction of The Institute of Electronics, Information and Communication Engineers, vol. E103-D, no. 02, pp. 348-362, February 2020. (Chapter 5)
- Yunosuke Higashi, Masao Ohira, Yutaro Kashiwa, and Yuki Manabe, “Hierarchical Clustering of OSS License Statements Toward Automatic Generation of License Rules,” Journal of Information Processing, vol. 27, pp. 42-50, January 2019.
- 柏 祐太郎, 大平 雅雄, 阿萬 裕久, 亀井 靖高, “大規模OSS開発における不具合修正時間の短縮化を目的としたバグトリージング手法,” 情報処理学会論文誌, vol. 56, no. 2, pp. 669-681, February 2015. (Chapter 5)

International Conferences (Refereed)

- Yutaro Kashiwa, “RAPTOR: Release-Aware and Prioritized Bug-fixing Task assignment Optimization,” In Proceedings of the 35th International Conference on Software Maintenance and Evolution (Doctoral Symposium), pp. 629-633, October 2019. (Chapter 5)
- Yutaro Kashiwa, Akinori Ihara, and Masao Ohira, “What Are the Perception Gaps between Floss Developers and SE Researchers?,” In Proceedings of the 15th International Conference on Open Source Systems, pp. 44-57, May 2019. (Chapter 3)
- Masao Ohira, Yutaro Kashiwa, Yosuke Yamatani, Hayato Yoshiyuki, Yoshiya Maeda, Nachai Limsettho, Keisuke Fujino, Hideaki Hata, Akinori Ihara, and Kenichi Matsumoto, “A Dataset of High Impact Bugs: Manually-Classified Issue Reports,” In Proceedings of the 12th Working Conference on Mining Software Repositories, pp. 518-521, May 2015. (Chapter 3)

-
- Yutaro Kashiwa, Hayato Yoshiyuki, Yusuke Kukita, and Masao Ohira, “A Pilot Study of Diversity in High Impact Bugs,” In Proceedings of the 30th International Conference on Software Maintenance and Evolution, pp. 536-540, September 2014. (Chapter 3)

International Conferences (Non-Refereed)

- Yutaro Kashiwa, Bram Adams, Akinori Ihara, and Masao Ohira, “Investigating the Impact of the Release Cycle on Bug Priority Prediction,” In Consortium for Software Engineering Research 2019 Spring Meeting, May 2019. (Chapter 4)
- Yutaro Kashiwa, Bram Adams, Akinori Ihara, and Masao Ohira, “Investigating the Impact of the Release Cycle on Bug Priority Prediction,” In Consortium for Software Engineering Research 2018 Fall Meeting, October 2018. (Chapter 4)

Domestic Conferences (Refereed)

- 柏 祐太郎, 山谷 陽亮, 大平 雅雄, “OSS 開発における不具合修正プロセスの改善に向けたモジュールオーナー候補者推薦,” 第24回ソフトウェア工学の基礎ワークショップ, pp. 165-170, 2017年11月. (Chapter 5)
- 久木田 雄亮, 柏 祐太郎, 大平 雅雄, “ソフトウェア開発状況の把握を目的とした変化点検出を用いたソフトウェアメトリクスの時系列データ分析,” ソフトウェア・シンポジウム 2015 in 和歌山 論文集, pp. 27-36, 2015年6月.
- 柏 祐太郎, 大平 雅雄, 阿萬 裕久, 亀井 靖高, “大規模OSS開発における不具合修正時間の短縮化を目的としたバグトリージ手法,” ソフトウェアエンジニアリングシンポジウム2014論文集, pp. 66-75, 2014年8月. (Chapter 5)
- 柏 祐太郎, 大平 雅雄, “不具合修正時間の短縮化を目的としたバグトリージ手法の提案,” 第20回ソフトウェア工学の基礎ワークショップ, pp. 053-058, 2013年11月. (Chapter 5)

Awards

- IPSJ Outstanding Paper Award, Transactions of Information Processing Society of Japan, 2015.
- Specially Selected Paper Award, Transactions of Information Processing Society of Japan, 2015.
- A Research Encouragement Award, Software Symposium in Wakayama, 2015. (the recipient: Yusuke Kukita)
- Student Research Award, IPSJ SIG Technical Reports SIGSE Special Interest Groups on Software Engineering, 2015. (the recipient: Yutaro Kashiwa)
- IPSJ Yamashita SIG Research Award, Software Engineering Symposium 2014, 2014. (the recipient: Yutaro Kashiwa)
- Best Paper Award, Software Engineering Symposium 2014, 2014.
- Interactive Award, Software Engineering Symposium 2013, 2013. (The recipient: Yutaro Kashiwa)

Bibliography

- [1] E. S. Raymond, *The Cathedral and the Bazaar*. California: O'Reilly Media, 1999.
- [2] J. Anvik and G. C. Murphy, "Reducing the Effort of Bug Report Triage: Recommenders for Development-Oriented Decisions," *ACM Transactions on Software Engineering and Methodology*, vol. 20, no. 3, pp. 1–35, 2011.
- [3] Y. Tian, D. Lo, and C. Sun, "DRONE: Predicting priority of reported bugs by multi-factor analysis," in *Proceedings of the 29th IEEE International Conference on Software Maintenance*, 2013, pp. 200–209.
- [4] Q. Mi, J. Keunga, Y. Huob, and S. Mensaha, "Not all bug reopens are negative: A case study on eclipse bug reports," *Information and Software Technology*, vol. 99, pp. 93–97, 2018.
- [5] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K. Matsumoto, "Predicting Re-opened Bugs: A Case Study on the Eclipse Project," in *Proceedings of the 17th Working Conference on Reverse Engineering*, 2010, pp. 249–258.
- [6] H. Hata, O. Mizuno, and T. Kikuno, "Bug Prediction Based on Fine-Grained Module Histories," in *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 200–210.
- [7] Y. Kamei, S. Matsumoto, A. Monden, K. Matsumoto, B. Adams, and A. E. Hassan, "Revisiting common bug prediction findings using effort-aware models," in *Proceedings of the IEEE International Conference on Software Maintenance*, 2010, pp. 1–10.
- [8] Z. Liu, Q. Huang, X. Xia, E. Shihab, D. Lo, and S. Li, "SATD detector: A text-mining-based self-Admitted technical debt detection tool," in *Proceedings of the International Conference on Software Engineering*, 2018, pp. 9–12.

- [9] D. Falessi, M. A. Shaw, F. Shull, K. Mullen, and M. Stein, “Practical considerations, challenges, and requirements of tool-support for managing technical debt,” in *Proceedings of the 2013 4th International Workshop on Managing Technical Debt*, 2013, pp. 16–19.
- [10] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. D. Penta, A. D. Lucia, and D. Shybyvanyk, “When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away),” *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1063–1088, 2017.
- [11] P. Bhattacharya and I. Neamtiu, “Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging,” in *Proceedings of 26th IEEE International Conference on Software Maintenance*, 2010, pp. 1–10.
- [12] G. Bortis and A. Van Der Hoek, “PorchLight: A tag-based approach to bug triaging,” in *Proceedings of the 35th International Conference on Software Engineering*, 2013, pp. 342–351.
- [13] G. Jeong, S. Kim, and T. Zimmermann, “Improving bug triage with bug tossing graphs,” in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The Foundations of Software Engineering*, 2009, pp. 111–120.
- [14] H. Kagdi, M. Gethers, D. Shybyvanyk, and M. Hammad, “Assigning change requests to software developers,” *Journal of software: Evolution and Process*, no. 1, pp. 3–33, 2012.
- [15] Z. Lin, F. Shu, Y. Yang, C. Hu, and Q. Wang, “An Empirical Study on Bug Assignment Automation Using Chinese Bug Data,” in *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement*, 2009, pp. 451–455.
- [16] D. Matter, A. Kuhn, and O. Nierstrasz, “Assigning bug reports using a vocabulary-

- based expertise model of developers,” in *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, 2009, pp. 131–140.
- [17] H. Naguib, N. Narayan, B. Brügge, and D. Helal, “Bug report assignee recommendation using activity profiles,” in *Proceedings of the 10th Working Conference on Mining Software Repositories*, 2013, pp. 22–30.
- [18] J. Park, M. Lee, J. Kim, S. Hwang, and S. Kim, “COSTRIAGE : A Cost-Aware Triage Algorithm for Bug Reporting Systems,” in *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, 2011, pp. 139–144.
- [19] M. M. Rahman, G. Ruhe, and T. Zimmermann, “Optimized Assignment of Developers for Fixing Bugs An Initial Evaluation for Eclipse Projects,” in *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement*, 2009, pp. 439–442.
- [20] R. Shokripour, Z. M. Kasirun, S. Zamani, and J. Anvik, “Automatic bug assignment using information extraction methods,” in *Proceedings of 2012 International Conference on Advanced Computer Science Applications and Technologies*, 2012, pp. 144–149.
- [21] A. Tamrawi, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen, “Fuzzy set and cache-based approach for bug triaging,” in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 365–375.
- [22] T. Zhang and B. Lee, “A hybrid bug triage algorithm for developer recommendation,” in *Proceedings of the ACM Symposium on Applied Computing*, 2013, pp. 1088–1094.
- [23] J. Park, M. Lee, J. Kim, S. Hwang, and S. Kim, “Cost-aware triage ranking algorithms for bug reporting systems,” *Knowledge and Information Systems*, vol. 48, no. 3, pp. 679–705, 2016.

- [24] L. An, F. Khomh, and Y. G. Guéhéneuc, “An empirical study of crash-inducing commits in Mozilla Firefox,” *Software Quality Journal*, vol. 26, no. 2, pp. 553–584, 2018.
- [25] Y. Kashiwa, A. Ihara, and M. Ohira, “What Are the Perception Gaps Between FLOSS Developers and SE Researchers? A Case of Bug Finding Research,” in *Proceedings of the 15th International Conference on Open Source Systems*, 2019, pp. 44–57.
- [26] R. K. Saha, J. Lawall, S. Khurshid, and D. E. Perry, “Are these bugs really ‘normal’?” in *Proceedings of the 12th IEEE International Working Conference on Mining Software Repositories*, 2015, pp. 258–268.
- [27] M. Gegick, P. Rotella, and T. Xie, “Identifying security bug reports via text mining: An industrial case study,” in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, 2010, pp. 11–20.
- [28] A. Nistor, T. Jiang, and L. Tan, “Discovering, reporting, and fixing performance bugs,” in *Proceedings of the 10th Working Conference on Mining Software Repositories*, 2013, pp. 237–246.
- [29] E. Shihab, A. Mockus, Y. Kamei, B. Adams, and A. E. Hassan, “High-impact defects: A study of breakage and surprise defects,” in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 300–310.
- [30] H. V. Garcia and E. Shihab, “Characterizing and Predicting Blocking Bugs in Open Source Projects Categories and Subject Descriptors,” in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 72–81.
- [31] T. Chen, M. Nagappan, E. Shihab, and A. E. Hassan, “An empirical study of dormant bugs,” in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 82–91.

- [32] M. Ohira, A. E. Hassan, N. Osawa, and K. Matsumoto, “The Impact of Bug Management Patterns on Bug Fixing : A Case Study of Eclipse Projects,” in *Proceedings of the 28th IEEE International Conference on Software Maintenance*, 2012, pp. 264–273.
- [33] M. Michlmayr and B. Fitzgerald, “Time-based release management in free and open source (FOSS) projects,” *International Journal of Open Source Software and Processes*, vol. 4, no. 1, pp. 1–19, 2012.
- [34] J. Teixeira, “Release Early, Release Often and Release on Time. An Empirical Case Study of Release Management,” in *Proceedings of the 13th International Conference on Open Source Systems*, 2017, pp. 167–181.
- [35] J. Anvik, L. Hiew, and G. C. Murphy, “Who should fix this bug?” in *Proceedings of the 28th International Conference on Software Engineering*, 2006, pp. 361–370.
- [36] J. Kanwal and O. Maqbool, “Bug prioritization to facilitate bug report triage,” *Journal of Computer Science and Technology*, vol. 27, no. 2, pp. 397–412, 2012.
- [37] D. Cubrani and G. C. Murphy, “Automatic bug triage using text categorization,” in *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*, 2004, pp. 92–97.
- [38] S. Lee, M. Heo, C. Lee, M. Kim, and G. Jeong, “Applying deep learning based automatic bug triager to industrial projects,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 926–931.
- [39] R. Shokripour, J. Anvik, Z. M. Kasirun, and S. Zamani, “A time-based approach to automatic bug report assignment,” *Journal of Systems and Software*, vol. 102, no. C, pp. 109–122, 2015.
- [40] K. Somasundaram and G. C. Murphy, “Automatic categorization of bug reports using latent Dirichlet allocation,” in *Proceedings of the 5th India Software Engineering Conference*, 2012, pp. 125–130.

- [41] W. Wu, W. Zhang, Y. Yang, and Q. Wang, “DREX: Developer recommendation with K-nearest-neighbor search and EXpertise ranking,” in *Proceedings of the 2011 18th Asia-Pacific Software Engineering Conference*, 2011, pp. 389–396.
- [42] X. Xia, D. Lo, X. Wang, and B. Zhou, “Dual analysis for recommending developers to resolve bugs,” *Journal of Software: Evolution and Process*, vol. 27, no. 3, pp. 195–220, 2015.
- [43] M. Linares-Vasquez, K. Hossen, H. Dang, H. Kagdi, M. Gethers, and D. Poshyvanyk, “Triaging incoming change requests: Bug or commit history, or code authorship?” in *Proceedings of the 2012 IEEE International Conference on Software Maintenance*, 2012, pp. 451–460.
- [44] F. Servant and J. A. Jones, “WHOSEFAULT: Automatic Developer-to-Fault Assignment through Fault Localization,” in *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 36–46.
- [45] G. John and P. Langley, “Estimating Continuous Distributions in Bayesian Classifiers,” in *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, 1995, pp. 338–345.
- [46] S. Gunn, “Support vector machines for classification and regression,” University of Southampton, Faculty of Engineering, Science and Mathematics; School of Electronics and Computer Science, Tech. Rep., 1998.
- [47] S. Wang, W. Zhang, and Q. Wang, “FixerCache: Unsupervised Caching Active Developers for Diverse Bug Triage,” in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2014, pp. 1–10.
- [48] N. Kumar Nagwani and S. Verma, “Rank-Me: A Java Tool for Ranking Team Members in Software Bug Repositories,” *Journal of Software Engineering and Applications*, vol. 05, no. 04, pp. 255–261, 2012.

- [49] M. Sharma, A. Tandon, M. Kumari, and V. B. Singh, “Reduction of Redundant Rules in Association Rule Mining-Based Bug Assignment,” *International Journal of Reliability, Quality and Safety Engineering*, vol. 24, no. 06, 2017.
- [50] I. Molyneaux, *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*, 1st ed. O’Reilly Media, Inc., 2009.
- [51] N. Nurmuliani, D. Zowghi, and S. Williams, “Using card sorting technique to classify requirements change,” in *Proceedings of 12th International Requirements Engineering Conference*, 2014, pp. 224–232.
- [52] Y. Kashiwa, H. Yoshiyuki, Y. Kukita, and M. Ohira, “A pilot study of diversity in high impact bugs,” in *Proceedings of the 30th International Conference on Software Maintenance and Evolution*, 2014, pp. 536–540.
- [53] P. Abate, R. Di Cosmo, L. Gesbert, F. Le Fessant, R. Treinen, and S. Zacchiroli, “Mining component repositories for installability issues,” in *Proceeding of the 12th Working Conference on Mining Software Repositories*, 2015, pp. 24–33.
- [54] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, “Programmers’ build errors: a case study (at google),” in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 724–734.
- [55] X. Zhao, X. Xia, P. S. Kochhar, D. Lo, and S. Li, “An empirical study of bugs in build process,” in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, 2014, pp. 1187–1189.
- [56] D. Felsing, S. Grebing, V. Klebanov, P. Rümmer, and M. Ulbrich, “Automating regression verification,” in *Proceedings of the 29th ACM/IEEE international conference on Automated Software Engineering*, 2014, pp. 349–360.
- [57] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, “Bug characteristics in open source software,” *Empirical Software Engineering*, vol. 19, no. 6, pp. 1665–1705, 2014.

- [58] S. H. Tan and A. Roychoudhury, “Relifix: Automated repair of software regressions,” in *Proceedings of the 37th International Conference on Software Engineering*, vol. 1, 2015, pp. 471–482.
- [59] V. T. Pham, W. B. Ng, K. Rubinov, and A. Roychoudhury, “Hercules: Reproducing crashes in real-world application binaries,” in *Proceedings of the 37th International Conference on Software Engineering*, vol. 1, 2015, pp. 891–901.
- [60] H. Seo and S. Kim, “Predicting recurring crash stacks,” in *Proceedings of the 27th International Conference on Automated Software Engineering*, 2012, pp. 180–189.
- [61] S. H. Tan, Z. Dong, X. Gao, and A. Roychoudhury, “Repairing Crashes in Android Apps,” in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 187–198.
- [62] M. Sullivan and R. Chillarege, “Software defects and their impact on system availability-a study of field failures in operating systems,” in *Proceeding of the 21st IEEE Fault Tolerant Computing Symposium*, 1991, pp. 2–9.
- [63] T. H. D. Nguyen, M. Nagappan, A. E. Hassan, M. Nasser, and P. Flora, “An industrial case study of automatically identifying performance regression-causes,” in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 232–241.
- [64] A. Nistor, P. C. Chang, C. Radoi, and S. Lu, “CARMEL: Detecting and fixing performance problems that have non-intrusive fixes,” in *Proceedings of the 37th International Conference on Software Engineering*, vol. 1, 2015, pp. 902–912.
- [65] S. Zaman, B. Adams, and A. E. Hassan, “Security Versus Performance Bugs: A Case Study on Firefox,” in *Proceedings of the 8th Working Conference on Mining Software Repositories*, 2011, pp. 93–102.
- [66] F. Gao, L. Wang, and X. Li, “BovInspector: automatic inspection and repair of buffer overflow vulnerabilities,” in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 786–791.

- [67] N. Meng, S. Nagy, D. Yao, W. Zhuang, and G. A. Argoty, “Secure Coding Practices in Java: Challenges and Vulnerabilities,” in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 372–383.
- [68] L. K. Shar, H. Beng Kuan Tan, and L. C. Briand, “Mining SQL injection and cross site scripting vulnerabilities using hybrid program analysis,” in *Proceedings of the 35th International Conference on Software Engineering*, 2013, pp. 642–651.
- [69] O. Kafali, J. Jones, M. Petruso, L. Williams, and M. P. Singh, “How Good Is a Security Policy against Real Breaches? A HIPAA Case Study,” in *Proceedings of the 39th International Conference on Software Engineering*, 2017, pp. 530–540.
- [70] B. Mathis, V. Avdiienko, E. O. Soremekun, M. Bohme, and A. Zeller, “Detecting information flow by mutating input data,” in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017, pp. 263–273.
- [71] J. P. Near and D. Jackson, “Finding security bugs in web applications using a catalog of access control patterns,” in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 947–958.
- [72] T. Menzies and A. Marcus, “Automated severity assessment of software defect reports,” in *Proceedings of the 2008 IEEE International Conference on Software Maintenance*, 2008, pp. 346–355.
- [73] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, “A large-scale empirical study of just-in-time quality assurance,” *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2013.
- [74] W. Abdelmoez, M. Kholief, and F. M. Elsalmy, “Bug Fix-Time Prediction Model Using Naïve Bayes Classifier,” in *Proceedings of the 2012 22nd International Conference on Computer Theory and Applications*, 2012, pp. 167–172.
- [75] G. Yang, S. Baek, J.-W. Lee, and B. Lee, “Analyzing emotion words to predict severity of software bugs,” in *Proceedings of the Symposium on Applied Computing*, 2017, pp. 1280–1287.

- [76] J. Ekanayake, J. Tappolet, H. C. Gall, and A. Bernstein, “Tracking Concept Drift of Software Projects Using Defect Prediction Quality,” in *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, 2009, pp. 51–60.
- [77] S. McIntosh and Y. Kamei, “Are Fix-Inducing Changes a Moving Target? A Longitudinal Case Study of Just-In-Time Defect Prediction,” *IEEE Transactions on Software Engineering*, vol. 44, no. 5, pp. 412–428, 2018.
- [78] A. Tsymbal, “The problem of concept drift: definitions and related work,” Department of Computer Science, Trinity College Dublin, Tech. Rep., 2004.
- [79] J. C. Schlimmer and R. H. Granger, “Incremental Learning from Noisy Data,” *Machine Learning*, vol. 1, no. 3, pp. 317–354, 1986.
- [80] G. Widmer and M. Kubat, “Learning in the presence of concept drift and hidden contexts,” *Machine Learning*, vol. 23, no. 1, pp. 69–101, 1996.
- [81] B. Adams and S. McIntosh, “Modern Release Engineering in a Nutshell – Why Researchers Should Care,” in *Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, 2016, pp. 78–90.
- [82] F. Khomh, T. Dhaliwal, Y. Zou, and B. Adams, “Do faster releases improve software quality? An empirical case study of Mozilla Firefox,” in *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, 2012, pp. 179–188.
- [83] M. Castelluccio, L. An, and F. Khomh, “Is it safe to uplift this patch? An empirical study on mozilla firefox,” in *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution*, 2017, pp. 411–421.
- [84] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, “How to break an API: cost negotiation and community values in three software ecosystems,” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 109–120.

- [85] S. Raemaekers, A. Van Deursen, and J. Visser, “Semantic versioning versus breaking changes: A study of the maven repository,” in *Proceedings of the 2014 14th IEEE International Working Conference on Source Code Analysis and Manipulation*, 2014, pp. 215–224.
- [86] F. Khomh, B. Adams, T. Dhaliwal, and Y. Zou, “Understanding the impact of rapid releases on software quality: The case of firefox,” *Empirical Software Engineering*, vol. 20, no. 2, pp. 336–373, 2015.
- [87] P. Kruchten, *Rational Unified Process, The: An Introduction*. Boston: Addison-Wesley, 2003.
- [88] P. Runeson, M. Alexandersson, and O. Nyholm, “Detection of duplicate defect reports using natural language processing,” in *Proceedings of the 29th International Conference on Software Engineering*, 2007, pp. 499–508.
- [89] J. Śliwerski, T. Zimmermann, and A. Zeller, “When Do Changes Induce Fixes?” in *Proceedings of the 2005 International Workshop on Mining Software Repositories*, 2005, pp. 1–5.
- [90] A. Bacchelli, M. D’Ambros, and M. Lanza, “Extracting source code from e-mails,” in *Proceedings of the 32nd IEEE International Conference on Program Comprehension*, 2010, pp. 24–33.
- [91] P. Moslehi, B. Adams, and J. Rilling, “Feature Location using Crowd-based Screen-casts,” in *Proceedings of the 15th IEEE Working Conference on Mining Software Repositories*, 2018, pp. 192–202.
- [92] S. W. Thomas, “Mining Unstructured Software Repositories Using IR Models,” Ph.D. dissertation, Queen’s University Kingston,, 2012.
- [93] S. Wang, L. L. Minku, and X. Yao, “A Systematic Study of Online Class Imbalance Learning With Concept Drift,” *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–20, 2018.

- [94] T. Saito and M. Rehmsmeier, “The precision-recall plot is more informative than the ROC plot when evaluating binary classifiers on imbalanced datasets,” *PLoS ONE*, vol. 10, no. 3, 3 2015.
- [95] D. Berrar and P. Flach, “Caveats and pitfalls of ROC analysis in clinical microarray research (and how to avoid them),” *Briefings in Bioinformatics*, vol. 13, no. 1, pp. 83–97, 2012.
- [96] C. O. Fritz, P. E. Morris, and J. J. Richler, “Effect size estimates: Current use, calculations, and interpretation,” *Journal of Experimental Psychology: General*, vol. 141, no. 1, pp. 2–18, 2 2012.
- [97] L. Kaufman and P. J. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis*. New York: John Wiley, 1990.
- [98] A. Lee, J. C. Carver, and A. Bosu, “Understanding the Impressions, Motivations, and Barriers of One Time Code Contributors to FLOSS Projects: A Survey,” in *Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering*, 2017, pp. 187–197.
- [99] S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations*. New York: John Wiley & Sons, Inc., 1995.
- [100] D. Pisinger, *Algorithms for Knapsack Problems*. Department of Computer Science, University of Copenhagen, 1995.
- [101] J. R. Quinlan, *C4.5: programs for machine learning*. San Francisco: Morgan Kaufmann Publishers Inc., 1993.
- [102] T. Hastie and R. Tibshirani, “Classification by Pairwise Coupling,” in *Advances in Neural Information Processing Systems*, M. I. Jordan, M. J. Kearns, and S. A. Solla, Eds., vol. 10. MIT Press, 1998.
- [103] J. Platt, *Fast Training of Support Vector Machines using Sequential Minimal Optimization*, B. Schoelkopf, C. Burges, and A. Smola, Eds. MIT Press, 1998.

- [104] Y. Kashiwa, M. Ohira, H. Aman, and Y. Kamei, “A BugTriaging Method for Reducing the Time to Fix Bugs in Large-scale Open Source Software Development,” *Journal of Information Processing Society of Japan*, vol. 56, No.2, pp. 669–781, 2015.
- [105] D. M. Blei, A. Y. Ng, and M. I. Jordan, “Latent Dirichlet Allocation,” *Journal of Machine Learning Research*, vol. 3, pp. 993–1022, 2003.
- [106] T. T. Nguyen, A. T. Nguyen, and T. N. Nguyen, “Topic-based, Time-aware Bug Assignment,” *ACM SIGSOFT Software Engineering Notes*, vol. 39, No.1, pp. 1–4, 2014.
- [107] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, “Indexing by Latent Semantic Analysis,” *Journal of the American Society of Information Science*, vol. 41, No.6, pp. 391–407, 1990.
- [108] T. Hofmann, “Probabilistic Latent Semantic Indexing,” in *Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 1999, pp. 50–57.
- [109] A. Mockus, R. T. Fielding, and J. D. Herbsleb, “Two case studies of open source software development: Apache and Mozilla,” *ACM Transactions on Software Engineering and Methodology*, vol. 11, no. 3, pp. 309–346, 2002.
- [110] C. Sun, D. Lo, X. Wang, J. Jiang, and S. Khoo, “A discriminative model approach for accurate duplicate bug report retrieval,” in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, 2010, pp. 45–54.
- [111] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, “An approach to detecting duplicate bug reports using natural language and execution information,” in *Proceedings of the 30th International Conference on Software Engineering*, 2008, pp. 461–470.
- [112] C. Bird, A. Gourley, P. Devanbu, A. Swaminathan, and G. Hsu, “Open Borders? Immigration in Open Source Projects,” in *Proceedings of the 4th International Workshop on Mining Software Repositories*, 2007, p. 6.

- [113] V. M. Mika, F. Khomh, B. Adams, E. Engstr, and K. Petersen, “On Rapid Releases and Software Testing,” in *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, 2013, pp. 20–29.
- [114] R. Arun, S. Vommina, C. V. Madhavan, and M. N. Murthy, “On Finding the Natural Number of Topics with Latent Dirichlet Allocation: Some Observations,” in *Proceedings of the 14th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining*, 2010, pp. 391–402.