Doctoral Dissertation

Understanding and Recommending Key Features to Improve Bug Management Process

Md. Rejaul Karim

September 6, 2019

Graduate School of Information Science
Nara Institute of Science and Technology

A Doctoral Dissertation
submitted to Graduate School of Information Science,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
Doctor of ENGINEERING

Md. Rejaul Karim

Understanding and Recommending Key Features to Improve Bug Management Process*

Md. Rejaul Karim

Abstract

Bug reports are the primary means through which developers triage and fix bugs. To achieve this effectively, bug reports need to be clearly described those features that are important for the developers. However, previous studies have found that reporters do not always provide such features that affect the different phases of the bug fixing process. In this dissertation, we first perform two empirical studies to investigate key features that reporters should provide in the description of the bug report to improve the bug-fixing process. We observe that (1) Steps to Reproduce, Test Case, Code Example, Stack Trace, and Expected Behavior are the key features that reporters often miss in their initial bug reports and developers require them for fixing bugs. (2) the degree of the key features varies among the different types of high impact bug reports, and (3) the additional requirement for the key features during bug fixing significantly affect the bug-fixing process. Then, we propose two approaches in order to support reporters to improve the bug fixing process. First, we develop classification models to predict whether reporters should provide certain key features in the description of bug reports by leveraging four popular machine-learning techniques. Then, we develop a key features recommendation model by leveraging historical bug-fixing knowledge and text mining techniques. We observe that (1) our models achieve promising F1-scores to predict key features; (2) Naïve Bayes Multinomial (NBM) outperforms other classification techniques to predict the key features based on the summary text of the bug reports; (3) our best performing model can work

*Doctoral Dissertation, Graduate School of Information Science, Nara Institute of Science and Technology, September 6, 2019.

successfully to predict key features in the cross-projects setting; and (4) our key features recommendation model can successfully recommend key features that reporters should provide in the description in bug reports. We believe that our findings and proposed models make a valuable contribution to improve the bug management process.

Keywords:

Bug Report, High Impact Bug, Open Source Project, Machine Learning

List of Major Publications

Early versions of the work in this thesis were published as listed below.

[1] Md. Rejaul Karim, Akinori Ihara, Eunjong Choi, and Hajimu Iida "Identifying and Predicting Key Features to Support Bug Reporting", Journal of Software: Evolution and Process (JSEP) (Accepted for the journal first track of International Conference on Software Maintenance and Evolution (ICSME 2019), Cleveland, Ohio, USA)

[2] Md. Rejaul Karim,"Key Features Recommendation to Improve Bug Reporting", 12th International Conference on Software and Systems Process (ICSSP 2019), Montreal, Canada.

[3] Md. Rejaul Karim, Akinori Ihara, Xin Yang, Hajimu Iida, and Kenichi Matsumoto "Understanding Key features to write High-Impact Bug Reports", 8th IEEE International Workshop on Empirical Software Engineering in Practice (IWESEP 2017), Tokyo, Japan.

[4] Md. Rejaul Karim, Akinori Ihara, Xin Yang, Hajimu Iida, and Kenichi Matsumoto. "What additional features should be included in High-impact Bug Report?" in the poster session of 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016), Osaka, Japan.

Other Related Publications

[1] Md. Rejaul Karim, Akinori Ihara, Xin Yang, Eunjong Choi, Hajimu Iida, and Kenichi Matsumoto, "Improving the High-Impact Bug Reports: A Case Study of Apache Project", IPSJ SIG Technical Reports, volume 2016-SE-192/2016-EMB-41, number 9, pages 1-7 June 2016.

[2] Md. Rejaul Karim, Akinori Ihara, Xin Yang, Hajimu Iida, and Kenichi Matsumoto. "An Approach for Improving the Quality of Bug Reports" in the poster session of MSR Asia Summit 2015, Kyoto, Japan.

Acknowledgements

I praise Allah the Creator, the Sustainer and the Evolver of the worlds, Lord of the heavens and of the earth and all that is between them, who has taught by the pen and has taught man that which be knew not, and who have adorned the lowest heaven with an ornament, the planets. I praise Him, the exalted Majestic, for guiding me to seek Science in Japan and giving me the power and the help to be patient perseverance to continuing my research journey.

I would like to express my gratitude to all those who have given the opportunity and supported me to complete this thesis.

First and foremost, I would like to show my sincere gratitude and deep appreciation to Professor Hajimu Iida, who grants me the priceless opportunity to study in the Laboratory for Software Design and Analysis. His enthusiasm to educate students and his very positive attitudes create a very friendly, healthy research environment in SDLab, in which I have been able to passionately conduct my research. His advice and guidance on my research also give me the confidence to continue in the field of mining software repository. Without his support, I would not successfully accomplish the doctoral and master degree and survive in Japan.

I would like to express deep appreciation to my co-supervisor, Lecturer Akinori Ihara for his continuous support and guidance in my research work. I am very much thankful and grateful to Akinori Ihara Sensei for his help. Without his valuable suggestions and guidance, I would not be able to successfully accomplish the doctoral and masters degree.

I would like to also have my deepest thanks to Assistant Professor Eunjong Choi. I have been feeling so grateful and fortunate to do research with her. Her continuous help for my research and her supportive attitude help me to successfully accomplish the doctoral and masters degree.

I would also like to thank my dissertation committee members: Professor

Kenichi Matsumoto and Associate Professor Kohei Ichikawa. Through expert critiques and suggestions, they helped me to shape this dissertation.

I would like to give special thanks to Xin Yang-san for his guidance and suggestions. His enthusiastic working also encourages me to work harder and want to become a good researcher.

I also want to thank all SDLab members for helping me to do research and live in Japan. Their passions in the study and their generous advice on my research encourage me to do better and work harder. Especially I want to express my appreciation to Kyohei Uemura-san, Erina Makihara-san, Char Karlong-san and Naoki Kawashima-san, Masaya Yamazaki-san for their support and guidance.

I would like to thanks Japanese Government for the generous financial support through the Monbukagakusho (MEXT) Scholarship. This scholarship enables me to live and pursue my research in Japan comfortably. It is a huge honor for me to be a recipient of this scholarship.

Finally and most importantly, I would like to thank my beloved mother and wife. Thank you for letting me follow my dreams and happily support me whatever I want to be. I would like to dedicate this dissertation to them as a small present of my gratitude for everything.

Dedication

To my parents

Contents

List of Figures

List of Tables

1 Introduction

One of the key activities in the software development process is fixing bugs reported by developers, testers, and end-users [1]. To fix these bugs, developers rely on the contents of bug reports [2]. A typical bug report contains input fields (e.g., a summary and description), which contain unstructured features such as what the reporters saw happen (Observed Behavior), what they expected to see happen (Expected Behavior), and a clear set of instructions that developers can use to reproduce the bugs (Steps to Reproduce). These unstructured features are crucial for the developers to triage and fix the bugs [3]. However, reporters often omit these features in their bug reports [4–7]. In addition, 78.1 percent of bug reports have a short description that contains fewer than 100 words [8]. Thus, developers often receive bug reports with a short description such as "Various minor edits"(Camel-4820) [9], "What is it" (Derby-893) [10], "See subject" (Wicket-1159) [11], and "See title" (Ambari-9083) [12]. The lack of unstructured features in bug reports is regarded as one of the key reasons for the amount of time taken to triage and fix bugs [4, 13] because developers have to spend much time and effort in order to understand the bugs based on features provided or need to ask reporters to provide additional features [4, 14, 15].

Well-described bug reports help in comprehending the problem, consequently increasing the likelihood of a bug being fixed [3]. Precise, well-described bug reports are more likely to gain the triager's attention [13]. Existing studies have proposed automated techniques to triage bugs, select appropriate developers, and localize bugs based on bug reports [16–20]. However, incomplete bug reports adversely affect the performance of these automated techniques [18]. Kim et. al. [20] found that almost half of all bug reports are unusable in terms of building a prediction model for localizing bugs. Hence, writing a well-described bug report is crucial to improving the bug-fixing process.

One of the main reasons for the lack of unstructured features in bug reports is inadequate tool support [1, 4, 21, 22]. In order to support reporters, researchers have focused on detecting the presence/absence of unstructured features in bug reports [21, 23, 24]. Zimmerman et al. [23] revealed 10 unstructured features that are important for developers in general. However, there is currently no consensus among software projects and bug reporting systems on the essential mandatory or optional unstructured features that should be part of a bug report [7]. Not all unstructured features are necessarily appropriate for all bug reports [5]. For example, Stack Trace is not generated for all bug reports. Previous empirical studies have found that bug reports contain only between two and six of the top 10 unstructured features [5, 14]. This indicates that not all unstructured features are equally important to fix all bugs. However, selecting those features that should be provided in a bug report is not easy for reporters, especially for novices and end-users. Thus, a tool that only detects the presence/absence of unstructured features is insufficient. Therefore, this study attempts to build an automated feature recommendation model to support reporters when writing new bug reports.

Bug-tracking systems for large open-source software (OSS) projects have more than 7,000 bug reports for each project. Thus, examining historical bug reports can be a good way to understand which features developers require to fix bugs. To better understand which features are important, we perform an exploratory study on five OSS projects using qualitative and quantitative analyses. In particular, we first perform a qualitative analysis to identify the key features by examining those that are provided initially (i.e., features that reporters provide in the initial bug reports), as well as additional required features (i.e., features that reporters missed in their initial submission, but that were required by developers to fix bugs). We manually investigate each bug report and identify the provided unstructured features from the initial bug reports. Then, we identify the additional required features that reporters provided in the comment sections after submitting the bug reports. In our previous study, we conducted a kick-off qualitative analysis of the Apache Camel project in order to better understand the key features of high-impact bug reports [14]. To generalize our findings, for now, we conduct a qualitative analysis of the Apache (Camel, Derby, and Wicket) and the Mozilla (Firefox and

Thunderbird) ecosystems' projects. Through qualitative and quantitative analyses, we identify five key features that reporters often miss in their initial bug reports and developers require them for fixing bugs.

Different types of bugs (e.g. Performance and Security bugs) vary from each other [25]. Therefore, our hypothesis is different types of high impact bug may need different key features to fix. To the best of our knowledge, there is no case study on revealing key features according to the high-impact bugs. Therefore, we motivated to do an empirical study on high-impact bug reports and to reveal key features so that we can improve the content of high impact bug reports. In order to investigate key features in terms of high impact bugs, we perform an empirical study on the Apache Camel project using qualitative and quantitative analyses. We manually investigate each bug report for each high impact bug and identify the provided unstructured features from the initial bug reports. Then, we identify the additional required features that reporters provided in the comment sections after submitting the bug reports. Through qualitative and quantitative analyses, we observe that the degree of key features varies among the different types of high impact bugs, however, four key features (Steps to Reproduce, Test Case, Code Example, Stack Trace) are almost same.

The summary is a mandatory field when writing a bug report. Here, reporters briefly describe the detected bug. The summary text has been used successfully to detect similar and duplicate bugs [26, 27]. By examining the contents of bug reports, we can determine which features are required to fix each bug. By applying machine-learning techniques, reporters of new bug reports know which key features need to provide based on the summary text by leveraging historical bug-fixing activities. Thus, in order to help reporters, in our quantitative analysis, we build prediction models using Naïve Bayes (NB), Naïve Bayes Multinomial (NBM), K-Nearest Neighbors (KNN), and Support Vector Machine (SVM) text-classification techniques, based on the summary text. Existing studies have found that the performance of prediction models varies between the text-classification techniques [28, 29] depending on the context. Hence, we use the aforementioned four popular text-classification techniques to build and compare prediction models. We evaluate our models using the bug reports of Camel, Derby, Wicket, Firefox, and Thunderbird projects. Our models achieve promising f1-scores when predicting key features, except for Stack Trace of the Wicket project and Code Example of Firefox project. Our comparative study of the classification techniques reveals that NBM outperforms the other techniques in terms of predicting key features. Finally, we build a key features recommendation model to suggest features that reporters should provide in the description to make a good bug report by providing the minimum number of features.

1.1 Thesis Contribution

We demonstrate that:

1. Steps to Reproduce, Test Case, Code Example, Stack Trace, and Expected Behavior are the five key features that reporters often miss in their initial bug reports and developers require them for fixing bugs. (Chapter 4)

2. The missing these key features in the initial bug reports significantly affect the bug fixing process. (Chapter 4)

3. The degree of key features vary among the different types of high impact bugs however, four key features (Steps to Reproduce, Test Case, Code Example, Stack Trace) are almost same. (Chapter 5)

4. Our models achieve the best f1-scores for Code Example, Test Case, Steps to Reproduce, Stack Trace, and Expected Behavior of 0.70 (Wicket), 0.70 (Derby), 0.70 (Firefox), 0.65 (Firefox), and 0.76 (Firefox), respectively, which are promising. (Chapter 6)

5. The Naïve Bayes Multinomial (NBM) outperforms the other techniques in terms of predicting key features. (Chapter 6)

6. Our best performing model can work for predicting the key features in the cross-project setting. (Chapter 6)

7. Our key features recommendation model can accurately recommend features by leveraging historical bug fixing knowledge. (Chapter 7)

1.2 Thesis Overview

In this section, we provide a brief overview of the thesis.

Chapter 2: Background and Theory: In this chapter, we provide a broad background and definitions of related terms and the steps of bug management process.

Chapter 3: Related Work: We present a survey and in-depth discussion of prior research that are related to this thesis.

Chapter 4: Understanding Key Features of a Bug Report: Prior research shows that there is a clear mismatch between the features that developers would wish to appear in a bug report, and the features that actually appear [3, 5] and bug reports are neither complete nor accurate, and often do not provide all the features that developers find useful when fixing bugs [5]. Yet, little is known about what features developers find useful during bug fixing. Hence, we perform an exploratory study on five OSS projects using qualitative and quantitative analyses. Through qualitative analysis, we identify five key features (i.e., Steps to Reproduce, Test Case, Code Example, Stack Trace, and Expected Behavior) that reporters often miss in their initial bug reports and developers require them for fixing bugs.

Chapter 5: Investigating Key Features of High Impact Bugs (HIB) Reports: Different types of bugs (e.g. Performance and Security bugs) vary from each other [25]. Therefore, our hypothesis is different types of high impact bug may need different key features to fix. However, there is no case study on revealing key features according to the high-impact bugs. Thus, we perform an empirical study on the high-impact bug reports of the Apache Camel project to reveal key features according to the high impact bugs.

Chapter 6: Predicting Key Features: One of the main reasons for the lack of unstructured features in bug reports is inadequate tool support [1, 4, 21, 22]. Thus, in order to help reporters, in our quantitative analysis, we build classification models using Naïve Bayes (NB), Naïve Bayes Multinomial (NBM), K-Nearest Neighbors (KNN), and Support Vector Machine (SVM) text-classification techniques, based on the summary text. We evaluate our models using the bug reports of Camel, Derby, Wicket, Firefox, and Thunderbird projects. Our models achieve promising f1-scores when predicting key features, except for the Stack Trace of the Wicket project and the Code Example of Firefox project.

Chapter 7: Key Features Recommendation Model: By examining the initial bug reports and bug fixing activities, we can determine which features are required to fix each bug. However, it is very difficult for reporters to know which features are required based on bug reports of fixed bugs from the repository without any automated techniques. Machine-learning techniques may help reporters to know which key features need to provide by leveraging historical bug-fixing activities. Hence, we propose a novel approach called key features recommendation model to suggest features that reporters should provide in the description of the bug reports. Our model utilizes the description and comments of the bug reports to generate a recommendation. We evaluate the accuracy of our recommendation model using the bug reports of three projects (Camel, Derby, and Wicket) from the Apache ecosystem and two projects (Firefox and Thunderbird) from the Mozilla ecosystem. The experimental results show that our model can accurately recommend key features.

1.3 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 provides background information, definition of key terms and techniques, and motivation of our study. Chapter 3 presents research related to our studies. Chapter 4 presents the findings of our qualitative and quantitative analyses to identify key features. Chapter 5 presents the findings of our case study on high impact bug reports. Chapters 6 presents and discusses the performance of our key features prediction models. Chapters 7 demonstrates our proposed key features recommendation model. Finally, Chapter 8 draws conclusion of our research.

2 Background and Theory

Bug fixing is one of the major activities in software maintenance to solve unexpected errors or crashes of software systems [30]. It is estimated that 80% of software development effort is spent on software maintenance [31]. A bug report plays an important role to triage and fix bugs. In this chapter, we explain a thorough detail background of our study. First section explains bug reporting process. Then, we discuss the different steps of bug fixing process and how a bug report helps to fix a bug in the second section. Third section discusses the motivation of this study.

2.1 Bug Reporting Process

Software bugs are expensive. Moreover, the cost of finding and fixing bugs represents one of the most expensive software development activities. Well-structured bug management process makes the life easy for the software practitioners to deal better way with software bugs. The figure 2.1 shows the conventional bug reporting process. In the conventional bug reporting process, a reporter creates a bug report. Then, the reporter submits the bug report to the bug tracking system. However, there is no intelligence checking system in the conventional bug reporting system whether the content of the bug report is sufficient. As a result, in many cases, reporters make insufficient bug reports.

In this section, first, we discuss what is a bug report and the content of a bug report. Then, we discuss the bugs in the bug tracking system.

2.1.1 Bug Report

A software bug is an error, flaw, failure or fault in a computer program or system that causes it to produce an incorrect or unexpected result, or to behave in unintended ways. Most bugs arise from mistakes and errors made by people in

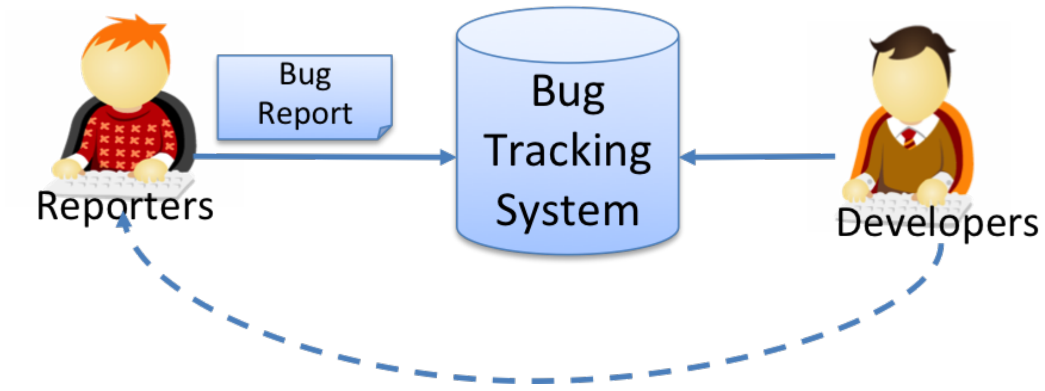


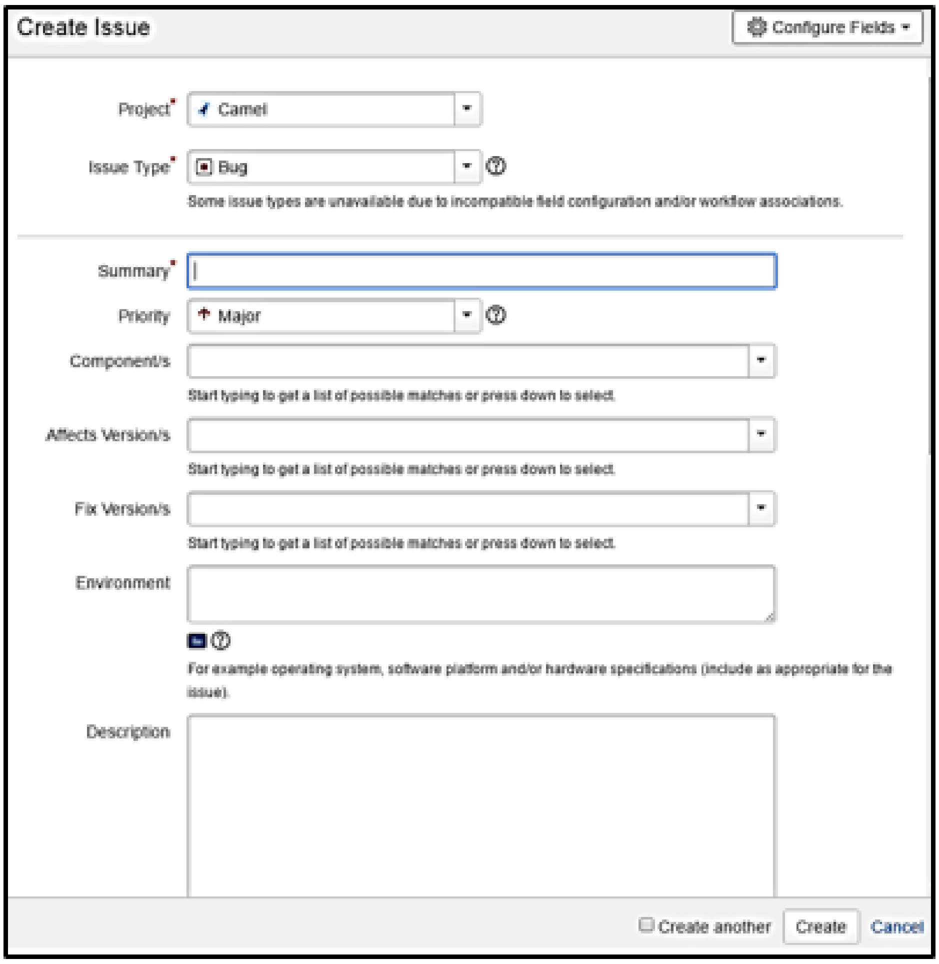
Figure 2.1: The conventional bug reporting process

either a program's source code or its design, or in frameworks and operating systems used by such programs, and a few are caused by compilers producing incorrect code. For example, software practitioners and end users face with different software problems like system crash, hang, produce an unexpected result, slower processing than expectation, unauthorized access into the system. All sort of problems are defined as software bug or defect. Software bugs are prevalent in all stages of the software development and maintenance lifecycle. When a bug identifies in a software project, the immediate next step is to create a good bug report. A report details about the identified bug is commonly known as bug reports or defect reports. The figure 2.2 depicts an example of a bug report creation form in Jira. The figure 2.3 depicts an example of a bug report that extracted from the Jira issue tracking system of the Apache camel project.

2.1.2 Content of a Bug report

The content of a bug report is a collection of structured and unstructured features. A feature is a pieces of information that describes a bug. The structured features usually represent the text defined by the project, such as the Component, Affected Version, and Priority. The unstructured features represent the text not defined by the project, such as Observed Behavior, Stack Trace, and Code Example.

Structured Features: A bug report contains many features. The structured features are those features that take only a limited number of values. The table


Create Issue

Configure Fields

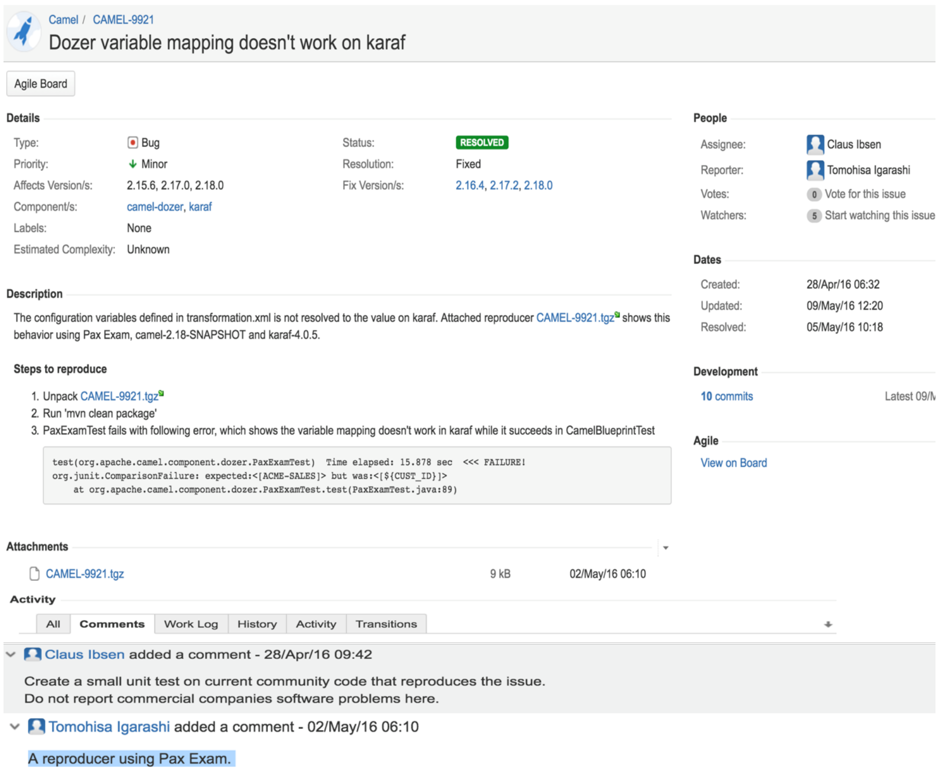
Project*	Camel
Issue Type*	Bug
	Some issue types are unavailable due to incompatible field configuration and/or workflow associations.
Summary*	
Priority	Major
Component/s	
	Start typing to get a list of possible matches or press down to select.
Affects Version/s	
	Start typing to get a list of possible matches or press down to select.
Fix Version/s	
	Start typing to get a list of possible matches or press down to select.
Environment	
	For example operating system, software platform and/or hardware specifications (include as appropriate for the issue).
Description	

Create another Create Cancel

Figure 2.2: An example of bug report creation form from Jira

2.1 shows the list of the structured features of a bug report.

Unstructured Features: The unstructured features are those features that describe using unstructured natural language text. The table 2.2 shows the list of unstructured features of a bug report that play important role to localize and to fix the bug.



Camel / CAMEL-9921

Dozer variable mapping doesn't work on karaf

Agile Board

Details

Type:	Bug	Status:	RESOLVED
Priority:	Minor	Resolution:	Fixed
Affects Version/s:	2.15.6, 2.17.0, 2.18.0	Fix Version/s:	2.16.4, 2.17.2, 2.18.0
Component/s:	camel-dozer, karaf		
Labels:	None		
Estimated Complexity:	Unknown		

People

Assignee:	Claus Ibsen
Reporter:	Tomohisa Igarashi
Votes:	0 Vote for this issue
Watchers:	5 Start watching this issue

Dates

Created:	28/Apr/16 06:32
Updated:	09/May/16 12:20
Resolved:	05/May/16 10:18

Development

10 commits Latest 09/M

Agile

View on Board

Description

The configuration variables defined in transformation.xml is not resolved to the value on karaf. Attached reproducer CAMEL-9921.tgz shows this behavior using Pax Exam, camel-2.18-SNAPSHOT and karaf-4.0.5.

Steps to reproduce

1. Unpack CAMEL-9921.tgz
2. Run 'mvn clean package'
3. PaxExamTest fails with following error, which shows the variable mapping doesn't work in karaf while it succeeds in CamelBlueprintTest

```
test(org.apache.camel.component.dozer.PaxExamTest)  Time elapsed: 15.878 sec  <<< FAILURE!
org.junit.ComparisonFailure: expected:<[ACME-SALES]> but was:<[${CUST_ID}]>
    at org.apache.camel.component.dozer.PaxExamTest.test(PaxExamTest.java:89)
```

Attachments

CAMEL-9921.tgz 9 kB 02/May/16 06:10

Activity

All | Comments | Work Log | History | Activity | Transitions

Claus Ibsen added a comment - 28/Apr/16 09:42

Create a small unit test on current community code that reproduces the issue.
Do not report commercial companies software problems here.

Tomohisa Igarashi added a comment - 02/May/16 06:10

A reproducer using Pax Exam.

Figure 2.3: An example of a typical bug report of the Apache Camel Project

2.1.3 Bug/Issue Tracking System

A bug tracking system or defect tracking system is a software application that keeps track of a reported software bug. Bugzilla [32] and Jira [33], are commonly use as a bug tracking systems to manage and facilitate the bug-fixing process in software development. It may also known as issue tracking system. A major component of a bug tracking system is a database that records meta data about the bugs. It may include the bug report submission time, its severity, the erroneous program behavior, and details on how to reproduce the bug; as well as the identity of the person who reported it and any programmers who may be working on fixing it.

Table 2.1: The list of structured features in a bug report

Name of Features	Short Description
Priority	The priority indicates how severe the bug is. Reporters assign different types of priority in Jira for a bug report such as major, minor, trivial. The general rule of thumb is the bug report with major priority should be fixed first
Environment	It indicates under which environment the bug is found eg., Windows XP, Linux
Affects Version	Name of version in which the bug is found
Component	Name of the module or package in where the bug is identified
Reporter	The person who files the bug report in the BTS. Reporter may be developer, tester, or end-user
Status	It indicates the current state of the bug report in the bug fixing process
Bug Resolver	The person who fixes the bug

2.2 Bug Fixing Process

A report plays a key role in sharing information about the bugs and the bug-fixing progress among developers. There are four basic steps in the bug-fixing process.

1. **Bug report submission.** A bug reporter describes an identified bug in a bug report. This report includes fields for providing a short summary and other features (e.g., Affected Version, Component, Observed Behavior, Steps to Reproduce, and Test Case) in the bug report. Then, the report is submitted to the BTS.

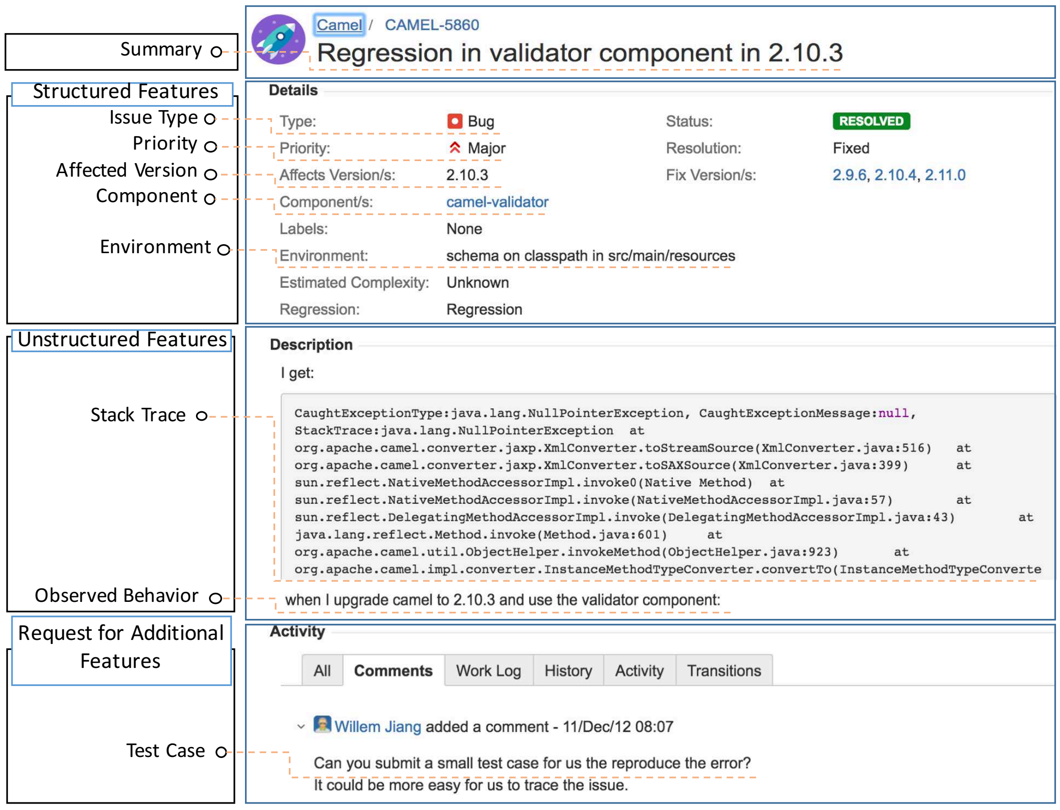
2. **Triaging and assignment.** A bug triager (e.g., project manager) decides whether the project should fix the bug because it may already have been submitted by other reporters (e.g., duplicate bug [34]). If the triager decides to fix the reported bug, he or she assigns it to an appropriate developer.

Table 2.2: The list of unstructured features in a bug report

Name of Features	Short Description
Summary	Summary should be a short and precise description of the bug. A good summary helps the developers to understand the bug quickly and uniquely. It should explain the problem, not your suggested solution.
Steps to Reproduce (STR)	A clear set of instructions that a developer can use to reproduce the bug on their own machine
Stack traces (ST)	A stack trace produced by the application, most often when the bug is reporting a crash in the application
Test Cases (TC)	One or more test cases that the developer can use to determine when they have fixed the bug
Observed behaviour (OB)	What the user saw happen in the application as a result of the bug
Screenshots (SS)	A screenshot of the application while the bug is occurring
Expected behaviour (EB)	What the user expected to happen, usually contrasted with Observed Behaviour
Code Examples (CE)	An example of some code which can cause the bug
Summary (S)	A short (usually one-sentence) summary of the bug
Version (V)	What version of the application the user was using at the time of the error
Error reports (ER)	An error report produced by the application as the bug occurred

3. **Localizing and fixing.** The assigned developer identifies the source code files that contain the reported bug. Here, the developer may request additional features, if needed. Then, the developer fixes the code.

4. **Verification.** A different developer (e.g., tester) verifies whether the corrected code now satisfies the reporter's requirements. If it does, then the developer closes the bug report. Otherwise, the bug report is reopened and



Camel / CAMEL-5860

Regression in validator component in 2.10.3

Details

Type:	Bug	Status:	RESOLVED
Priority:	Major	Resolution:	Fixed
Affects Version/s:	2.10.3	Fix Version/s:	2.9.6, 2.10.4, 2.11.0
Component/s:	camel-validator		
Labels:	None		
Environment:	schema on classpath in src/main/resources		
Estimated Complexity:	Unknown		
Regression:	Regression		

Description

I get:

```
CaughtExceptionType:java.lang.NullPointerException, CaughtExceptionMessage:null,
StackTrace:java.lang.NullPointerException  at
org.apache.camel.converter.jaxp.XmlConverter.toStreamSource(XmlConverter.java:516)   at
org.apache.camel.converter.jaxp.XmlConverter.toSAXSource(XmlConverter.java:399)      at
sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)  at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)        at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)        at
java.lang.reflect.Method.invoke(Method.java:601)     at
org.apache.camel.util.ObjectHelper.invokeMethod(ObjectHelper.java:923)       at
org.apache.camel.impl.converter.InstanceMethodTypeConverter.convertTo(InstanceMethodTypeConverte
```

when I upgrade camel to 2.10.3 and use the validator component:

Activity

Willem Jiang added a comment - 11/Dec/12 08:07

Can you submit a small test case for us the reproduce the error?
It could be more easy for us to trace the issue.

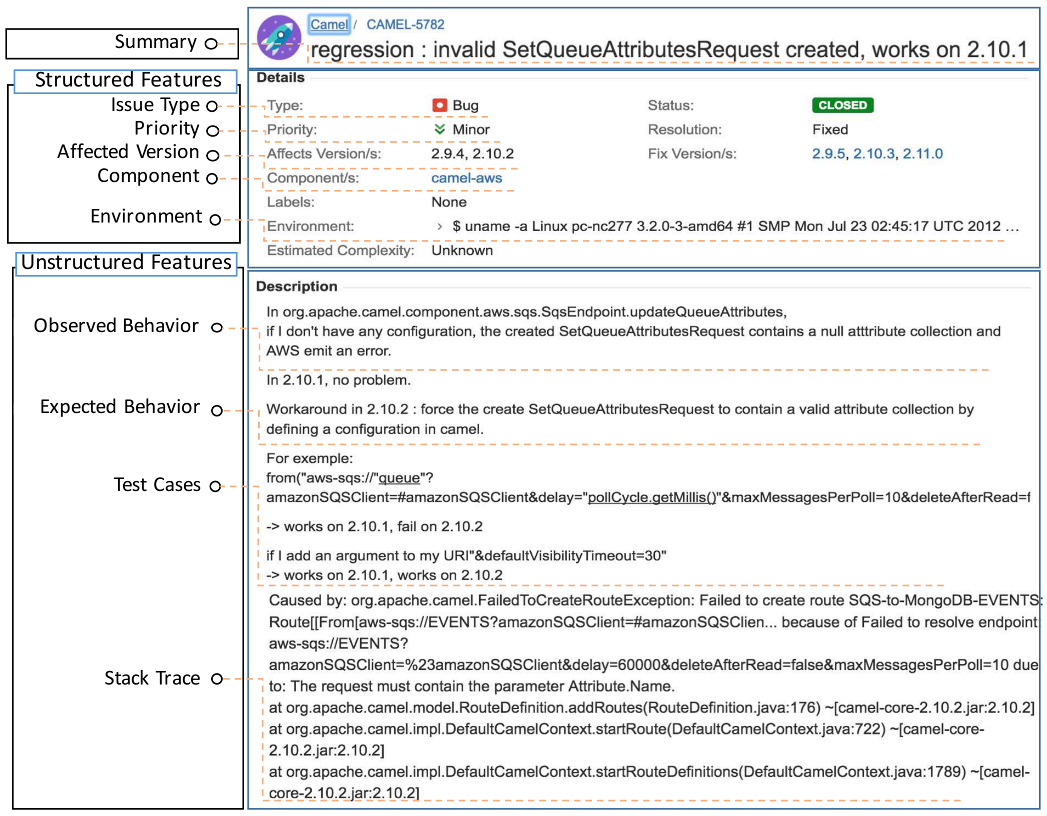
Figure 2.4: Example of a bug report that required additional features to fix the bug

the triager reassigns it to the developer for correction.

The bug report plays an important role in fixing a bug in all steps of the bug-fixing process.

2.3 Motivation for Study

Existing studies [3, 5, 14] have shown that developers rely on bug reports when fixing bugs. However, it is sometimes difficult to reproduce and localize these bugs. For example, Figure 2.4 shows an example of a bug report [35] where the developer was required additional features about the detected bug in the Camel project. The reporter, David J.M. Karlsen, provided Observed Behavior and Stack



Camel / CAMEL-5782

regression : invalid SetQueueAttributesRequest created, works on 2.10.1

Details

Type:	Bug	Status:	CLOSED
Priority:	Minor	Resolution:	Fixed
Affects Version/s:	2.9.4, 2.10.2	Fix Version/s:	2.9.5, 2.10.3, 2.11.0
Component/s:	camel-aws		
Labels:	None		
Environment:	› $ uname -a Linux pc-nc277 3.2.0-3-amd64 #1 SMP Mon Jul 23 02:45:17 UTC 2012 …		
Estimated Complexity:	Unknown		

Description

In org.apache.camel.component.aws.sqs.SqsEndpoint.updateQueueAttributes,
if I don't have any configuration, the created SetQueueAttributesRequest contains a null atttribute collection and AWS emit an error.

In 2.10.1, no problem.

Workaround in 2.10.2 : force the create SetQueueAttributesRequest to contain a valid attribute collection by defining a configuration in camel.

For exemple:
from("aws-sqs://"queue"?
amazonSQSClient=#amazonSQSClient&delay="pollCycle.getMillis()"&maxMessagesPerPoll=10&deleteAfterRead=f

-> works on 2.10.1, fail on 2.10.2

if I add an argument to my URI"&defaultVisibilityTimeout=30"
-> works on 2.10.1, works on 2.10.2

```
Caused by: org.apache.camel.FailedToCreateRouteException: Failed to create route SQS-to-MongoDB-EVENTS:
Route[[From[aws-sqs://EVENTS?amazonSQSClient=#amazonSQSClien... because of Failed to resolve endpoint:
aws-sqs://EVENTS?
amazonSQSClient=%23amazonSQSClient&delay=60000&deleteAfterRead=false&maxMessagesPerPoll=10 due
to: The request must contain the parameter Attribute.Name.
at org.apache.camel.model.RouteDefinition.addRoutes(RouteDefinition.java:176) ~[camel-core-2.10.2.jar:2.10.2]
at org.apache.camel.impl.DefaultCamelContext.startRoute(DefaultCamelContext.java:722) ~[camel-core-
2.10.2.jar:2.10.2]
at org.apache.camel.impl.DefaultCamelContext.startRouteDefinitions(DefaultCamelContext.java:1789) ~[camel-
core-2.10.2.jar:2.10.2]
```

Figure 2.5: Example of a bug report without a request for additional features

Trace as the unstructured features in the description of the bug report. When the developer, Willem Jiang, received the assignment to fix the bug, he failed to reproduce the bug and asked for a Test Case from the reporter one day later. The reporter then responded 77 days later. Such delays can be costly in terms of performance and may influence reports of new bugs. To avoid these kinds of delay, we propose an approach that suggests which unstructured features developers will require in order to fix the bugs. The approach will be especially helpful for novices and end-users when writing a new bug report.

Figure 2.5 shows another example of a bug report [36] from the Apache Camel project. This bug was reported and fixed one month earlier than that shown in Fiqure 2.4. The reporter, Julien Graglia, provided the similar structured features as the first example. However, he clearly mentioned Observed Behavior, Expected

Behavior, Test Cases, and Stack Trace as unstructured features in the description of the bug report. The developer, Christian Muller, found sufficient features in the bug report to localize and fix the bug. Finally, the developer fixed the bug within one day.

In both examples, the bug reports are for regression-related bugs and both contain almost similar structured features except priority. Although the priority of the bug report CAMEL-5860 was “Major”, the developer took a long time to fix the bug. On the other hand, the priority of the bug report CAMEL-5782 was “Minor”, the developer fixed within one day after the assignment. Usually, a bug report with the higher priority gets more attention than the lower priority by the developers because of its impact on the software project. After analyzing the bug-fixing activities, we find that the Test Case is essential in both cases. However, the reporter of the bug report CAMEL-5860 (see Fiqure 2.4) did not provide this feature in the initial submission, which delayed fixing the bug, in contrast to the bug report CAMEL-5782 (see Fiqure 2.5). The first report, CAMEL-5860, was created one month after the bug in CAMEL-5782 was fixed. The reporter of CAMEL-5860 might have known which features are essential to fixing the bug because a similar type of bug in CAMEL-5782 was already fixed. Thus, the reporter could have saved valuable time for the developers by providing these features in the initial submission. However, it is very difficult for reporters to know what features are required based on fixed bug reports from the repository without any automated techniques. This motivates us to develop classification models to predict the key features reporters should provide in initial bug reports based on reports of bugs that have already been fixed.

2.4 Chapter Summary

This chapter provides a broad background of bug reporting and bug fixing process, and describe different types of structured and unstructured features of a bug report that helps to describe a software bug. Then, we describe our motivation with two practical examples to conduct this research. In the next chapter, we survey prior research on bug reporting in order to situate our studies with respect to the literature.

3 Related Work

In this chapter, we survey the related research on improving bug management process. More specifically, we describe how the related work motivates our empirical studies and building models to support bug reporting.

3.1 Contents of a Bug report

Many empirical studies have proposed ways in which to improve the contents of bug reports. Bettenburg et al. [3] conducted a survey of 156 experienced developers and reporters from three OSS projects to examine what features developers expect to see in bug reports. As a result of their survey, they revealed 16 important structured and unstructured features for fixing bugs. They also developed a prototype tool called "CUEZILLA" to measure the quality of bug reports. To validate their prototype tool, they randomly selected 289 bug reports and then asked developers to assess their quality on a five-point Likert scale ranging from very poor to very good [37]. Then, they used these 289 bug reports to train and evaluate CUEZILLA by building supervised learning models. Their models achieved 45% accuracy when measuring the quality of the bug reports. In contrast, we build prediction models based on the titles/summaries of bug reports to notify reporters on which features to provide in a new bug report.

Davies et al. [5] conducted a case study on four OSS projects based on the top 10 important features (see Table 4.2) to understand which features reporters provide in bug reports. They found that bug reports do not always provide all important features. Furthermore, they found that 12% of all of features are provided after the initial submissions of the bug reports. As a result, developers spend valuable time collecting the required features. In order to inform the design of new bug-reporting tools, Ko et al. [38] conducted a linguistic analysis of the titles of bug reports. They observed a large degree of regularity and a substantial number of references to visible software entities, physical devices, or user actions. Their results suggest that future BTSs should collect data in a more structured way. Our study focuses on revealing the additional required features that reporters often omit from bug reports. These additional requirements during bug fixing might increase the time required to fix the bugs. Thus, to reduce the additional requirements and to improve the contents of bug reports, we build prediction models using popular text-mining techniques (i.e., NB, NBM, KNN, and SVM).

Chaparro et al. [21] conducted an empirical study to understand the extent to which Observed Behavior, Steps to Reproduce, and Expected Behavior are reported in bug reports and what discourse patterns (i.e., rules that capture the syntax and semantics of the text e.g., "To reproduce" and "STR" are the discourse patterns of Steps to Reproduce) reporters use to describe such information. Then, they designed an automated approach to detect the absence or presence of Steps to Reproduce and Expected Behavior in bug descriptions. Their approach intends to warn reporters if they forget to provide these features in the descriptions. In contrast, our approach helps reporters to understand which features should be provided in the descriptions when writing bug reports. Thus, reporters can create effective bug reports by providing the minimum number of features.

In addition, many existing studies use the features of bug reports to improve the bug-fixing process. Hooimeijer et al. [39] presented a basic linear regression model that predicts whether bug reports are resolved within a given period. Their model is based on structured features (i.e., Daily Load, Submitter Reputation, Readability, and Severity) that can be readily extracted from a bug report within a day of its initial submission to the repository. In contrast, our models are based on unstructured features (e.g., Summary, Steps to Reproduce, Test Case, and Code Example). Another important difference is that our models intend to recommend which features reporters should provide in the description to create a good bug report. Several other studies have applied automated techniques to select appropriate developers [16], localize bugs [18] and predict bug-fixing effort [19].

These works motivate us to conduct an exploratory study to understand the key features by examining the bug reports of OSS projects and developers' activities during bug fixing. Then, we build prediction models by leveraging popular text-mining techniques that enable reporters to know which key features are required by developers during bug fixing.

3.2 High Impact Bug (HIB)

Although we did not find any studies that directly research the topic of key features that play important role to fix high-impact bugs, we did find several studies that are related to our works.

Existing studies defined six types of HIB and they found that HIB should be fixed quicker than other bugs in software development [40] [41] [1] [42] [43] [44]. For example, security bug (one of the type of HIB) should be fixed faster than other non-HIB because it allows unauthorized access to the system. However, fixing a HIB sometimes become complicated because of the low-quality bug reports. Davies et al. found that bug reports are neither complete nor accurate through a case study on four big scale and successful open source projects (Eclipse, Firefox, Apache HTTP, and Facebook API) [5]. Thus, an empirical study to identify key features according to high-impact bugs is essential.

In a large and evolving software system, the large amount of bug reports typically exceed the available project resources. Accordingly, some bugs might be dealt with a long term delay or not at all [39]. Guo et al. [45] found that one of the main reasons for the reassigned bug reports in multiple time because of insufficient information in the description of bug reports. It is very difficult to understand and assign appropriate developers if the bug reports are not well-written. Developers also loose interest to fix bugs because of too long description of bugs [8, 45]. Thus, a bug report with necessary features are crucial for the developers to fix bugs accurately.

Shahed Zaman et al. [25] conducted a case study on Firefox project and studied how performance differ from security bugs in a software project. Authors found that security bugs are fixed and triaged much faster, but are reopened and tossed more frequently. Authors considered bug resolution time, triaging time, no. of reopen bugs, developers experiences, no. of files changes etc. to make comparison between performance and security bugs. Thus, our hypothesis is developers may need different features for fixing high impact bugs. In this research, we manually investigate six types of high-impact bugs to identify key features by examining bug fixing activities.

Several studies that applied automatic techniques to categorize and localize high impact bug [28, 29, 41–43, 46, 47]. All these approaches should benefit by our case study because, the performance of their model affect on the quality of the bug report. These works motivate us to conduct an empirical study to understand the key features of each type of HIBs.

3.3 Tools to Support Bug Reporting

In OSS projects, most bug reporters work voluntarily. In many cases, reporters submit incomplete or inaccurate bug reports that affect the bug fixing process. Thus, tools support is essential to improve the bug fixing process. In an effort to support bug reporting, Zimmermann et al. [1] and Chaparro et al. [21] develop prototype tools. Their tools can detect missing features in the description of the bug report and notify the reporters. However, not all unstructured features are necessarily appropriate for all bug reports [5]. For example, Stack Trace is not generated for all bug reports. Previous empirical studies have found that bug reports contain only between two and six of the top 10 unstructured features [5, 14]. This indicates that not all unstructured features are equally important to fix all bugs. Thus, existing tools provide unnecessary notifications to the reporters that might increase the complexity of bug reporting. Joel Spolsky noted that *"I have always felt that if you can make it 10 percent easier to fill in a bug report, you will get twice as many bug reports"* [48]. Thus, more appropriate tool support is essential to improve the content of bug reports.

Bug-tracking systems for large open-source software (OSS) projects contain a large number of bug reports of fixed bugs. For each bug, it contains initial bug reports, developers and reporters activities during fixing, patches that fixed bugs, etc. Thus, examining historical bug reports can be a good way to understand which features developers require to fix bugs. However, it is very difficult for reporters to know what features are required based on fixed bug reports from the repository without any automated techniques. In recent years, information retrieval (IR) and machine learning based automatic techniques has gained popularity to identify similar bugs [28, 49, 50] and duplicate bugs [34, 51, 52]. Rocha et al. [49] used the summary text of the bug reports to train their model and perform well to identify pending bug reports. Some other researchers also use IR based techniques to recommend similar bug reports [53–55]. In our approach, we also used the summary text of the bug reports to identify the bug reports of similar bugs. The term frequency-inverse document frequency (tf-idf) and document similarity score are widely used in software engineering to identify similar documents [56–61]. To build the vector space model, we use the tf-idf term weighting techniques so that important terms get a higher weight than others. We apply cosine similarity technique to calculate the similarity score of the new bug reports with the bug reports of the fixed bugs. Prior researchers widely use TopN (N=1,2,3,......N, means top N similar documents) similarity score [62–67] to recommend appropriate developers and similar, duplicate bug reports. In our research, we also use similar techniques to select the most similar bug reports.

3.4 Chapter Summary

In this chapter, we survey prior research along the features to write a bug report, and existing tools to support reporters for making a good bug report. From the survey, we find that (1) little is known about the key features that make a good bug report (2) there is lack of tools support for the reporters to make a good bug report.

Broadly speaking, the remainder of this thesis describes our empirical studies that identify the key features, which developers find useful during bug fixing (Chapters 4 and 5), and novel approaches to support bug reporting in order to improve the contents of the bug reports (Chapters 6 and 7).

4 Understanding Key Features of a Bug Report

The goal of this chapter is to perform an exploratory study on bug reports to understand key features to write a good bug report. To achieve this goal, we perform a qualitative analysis to investigate the features of bug reports by examining those that are provided initially (i.e., features that reporters provide in the initial bug reports), as well as additional required features (i.e., features that reporters missed in their initial submission, but that were required by developers to fix bugs). Through a case study of three projects (Camel, Derby, and Wicket) from Apache ecosystem and two projects (Firefox and Thunderbird) from Mozilla ecosystem, we identify five key features that developers often request during bug fixing. We also find that missing these key features in the initial bug reports significantly affect the bug fixing process.

4.1 Introduction

One of the key activities in the software development process is fixing bugs reported by developers, testers, and end-users [1]. To fix these bugs, developers rely on the contents of bug reports [2]. A typical bug report contains input fields (e.g., a summary and description), which contain unstructured features such as what the reporters saw happen (Observed Behavior), what they expected to see happen (Expected Behavior), and a clear set of instructions that developers can use to reproduce the bugs (Steps to Reproduce). These unstructured features are crucial for the developers to triage and fix the bugs [3]. However, reporters often omit these features in their bug reports [4–7]. In addition, 78.1 percent of bug reports have a short description that contains fewer than 100 words [8]. Thus, developers often receive bug reports with a short description such as "Various minor edits" (Camel-4820), "What is it" (Derby-893), "See subject" (Wicket-1159), and "See title" (Ambari-9083). The lack of unstructured features in bug reports is regarded as one of the key reasons for the amount of time taken to triage and fix bugs [4,13] because developers have to spend much time and effort in order to understand the bugs based on features provided or need to ask reporters to provide additional features [4,14,15]. Well-described bug reports help in comprehending the problem, consequently increasing the likelihood of a bug being fixed [3]. Precise, well-described bug reports are more likely to gain the triager's attention [13]. Existing studies have proposed automated techniques to triage bugs, select appropriate developers, and localize bugs based on bug reports [16–20]. However, incomplete bug reports adversely affect the performance of these automated techniques [18]. Kim et. al. [20] found that almost half of all bug reports are unusable in terms of building a prediction model for localizing bugs. Hence, writing a well-described bug report is crucial to improving the bug-fixing process.

Bug-tracking systems for large open-source software (OSS) projects have more than 7,000 bug reports for each project. Thus, examining historical bug reports can be a good way to understand which features developers require to fix bugs. To better understand which features are important, we perform an exploratory study on five OSS projects using qualitative and quantitative analyses. In particular, we first perform a qualitative analysis to identify the key features by examining those that are provided initially (i.e., features that reporters provide in the initial bug reports), as well as additional required features (i.e., features that reporters missed in their initial submission, but that were required by developers to fix bugs). We manually investigate each bug report and identify the provided unstructured features from the initial bug reports. Then, we identify the additional required features that reporters provided in the comment sections after submitting the bug reports. Through qualitative analysis, we identify five key features that reporters often miss in their initial bug reports and developers require them for fixing bugs. Our contributions are two-fold:

- **We perform an exploratory study on five OSS projects to investigate the initially provided and additional required features.** We identify the initially provided features from the submitted bug reports and additional required features during bug-fixing. Through our qualitative analysis, we identify three features of the nine important features (i.e., Observed Behaviour, Expected Behaviour, and Code Example) that reporters frequently provide (other than the summary). Then, the most common additional features required during bug fixing are Steps to Reproduce, Test Case, Code Example, Stack Trace, and Expected Behavior.

- **We perform a quantitative analysis to understand the impact of the additionally requested features on the bug fixing process**. To determine the impact, we divide the bug reports into two groups (i.e., bug reports with additional required features and those without additional required features) for each project. Then, we analyze the bug-fixing time, the number of comments made by developers during bug fixing, and the number of commentators who participated in the bug-fixing process for each group. Then, we perform some statistical analysis of both groups. Our findings suggest that the additional required features have a significant impact on the bug-fixing process.

4.2 Case Study Design

The lack of important features in bug reports is one of the main reasons for non-reproduced bugs [15], unfixed bugs [1], and additional bug triage effort [4], as developers have to spend more time and effort understanding bug descriptions or asking for clarifications and additional features [4, 15]. Low-quality bug reports are also likely to gain low attention by developers [13]. As indicated by developers, absent and wrong features in bug reports is the predominant cause for delays on bug fixing [1]. One of the reasons for submitting incomplete or less informative bug reports is lack of proper knowledge gap about the key features that are important for the developer to fix bugs.

In this research, we conduct an exploratory study on bug reports of OSS projects to understand how to write a good bug report by analyzing historical bug reports.We design our analysis to reduce the knowledge gap of the reporters especially novice users and end-users. In our analysis, we consider bug reports of fixed bugs. Usually, developers try to fix the bugs based on the initially provided features in the bug reports. If the develops require features that are missing in the initial bug reports then they request to provide those features additionally during bug fixing. Thus, our assumption is bug reports of fixed bugs contain features that are important for the develops to fix the bugs. We can reveal the key features to write a good bug report by analyzing the initial bug reports and bug fixing activities. To do so, we extract fixed bug reports (i.e., marked as FIXED) from JIRA (Camel, Derby, and Wicket projects) and BugZilla (Firefox and Thunderbird projects) repositories. We then conduct a qualitative analysis to identify the key features to write a good bug report. We first manually examine each bug report and identify the initially provided features. Then, we examine bug-fixing activities, especially the comments section, and identify additional required features. Finally, we conduct a statistical analysis to understand the impact of the additional features on the bug-fixing process. Figure 4.1 provides an overview of our study. Our study comprises two phases: P1: Data Set Preparation (DP); P2: Qualitative and Quantitative Analysis (QQA). We describe each phase below.

4.2.1 Datasets Preparation

In order to prepare data sets, we first set up three essential criteria for selecting target projects. Then, we generate the sample size and randomly select bug reports to prepare the sample data set for each of the selected projects.

- **Criterion 1 - Projects have a large number of bug reports in the issue-tracking system.** A previous study [40] found that a data set containing few bug reports is difficult to use when building data-mining or machine-learning models. Thus, we target projects with a large number of bug reports because this indicates that the project is more mature and stable.
- **Criterion 2 - Projects have well-structured bug-fixing histories.** This study analyzes the historical communication logs between developers and reporters to understand what features were required to fix each bug.
- **Criterion 3 - Projects differ in terms of their application domains.** The contents of bug reports may vary between different application domains.

To increase the generalizability of our results, we need to select projects from different application domains for our study.

We initially selected four projects from the Apache Software Foundation (ASF)[1] and two projects from the Mozilla Foundation that met the above criteria. The projects are Ambari[2], Camel[3], Derby[4], Wicket[5], Firefox[6], and Thunderbird[7]. However, we found that the majority of the bug reports in the Ambari project are self-reported (i.e., reporters fix the detected bugs themselves). Self-reported bug reports have an impact on the bug-fixing process [68]. Because we conjecture that self-reported bug reports are likely to have incomplete descriptions, we exclude the Ambari project from our case study. The following is the brief description of each of the selected projects.

Apache Camel: An open source framework for message-oriented middleware with a rule-based routing and mediation engine, written in Java.

Apache Derby: A relational database management system developed, written in Java.

Apache Wicket: A component-based web application framework for the Java programming language, written in Java.

Mozilla Firefox: A free and open-source web browser, written mainly in C++.

Mozilla Thunderbird: A free and open-source cross-platform email client, written mainly in C++.

The JIRA and BugZilla contains a large number of bug reports for each of the selected projects. To make our manual analysis simple and rational, we generate a statistically representative sample size for each project. To obtain proportion estimates that are within 5% bounds of the actual proportion, with a 95% confidence level, we randomly select a sample of size $s = \frac{z^2 p(1-p)}{0.05^2}$, where p is the proportion we want to estimate and $z = 1.96$. Because we do not know the proportion in advance, we use p = 0.5. We further correct for the finite population

[1]https://www.apache.org/
[2]https://issues.apache.org/jira/projects/AMBARI
[3]https://issues.apache.org/jira/projects/CAMEL
[4]https://issues.apache.org/jira/projects/DERBY
[5]https://issues.apache.org/jira/projects/WICKET
[6]https://bugzilla.mozilla.org/buglist.cgi?quicksearch=Firefox
[7]https://bugzilla.mozilla.org/buglist.cgi?quicksearch=Thunderbird

Table 4.1: Statistics of bug reports and the sample size for each target project.

Ecosystems	Projects	#Total Issue Reports	#Fixed Bug Reports	#Sample Size
Apache	Camel	11798	3964	350
Apache	Derby	6955	4063	351
Apache	Wicket	6466	3946	350
Mozilla	Firefox (General)	10000	5353	358
Mozilla	Thunderbird (General)	8693	1614	310

of bug reports P using $ss = \frac{s}{1+\frac{s-1}{P}}$ to obtain the sample for our qualitative analysis. Table 4.1 shows the statistics for the analyzed bug reports.

In order to prepare the data set for each target project according to the sample size, we first filter out bug reports that satisfy criterion i, described below. Then, we randomly select bug reports from the set of all reports according to the sample size. In our analysis, we also exclude bug reports that satisfy criterion ii and replace them with other, randomly selected bug reports.

- **Criterion i - The bug reporter and the fixer is the same person.** If the bug reporter and the bug fixer is the same person, then the report may not include all of the features required to fix the bugs. Therefore, we exclude these bug reports.
- **Criterion ii - The provided URL no longer exists.** Some bug reporters provide a URL of a website instead of writing features in the description. At the time of our analysis, we could not access some of those URLs. Therefore, we exclude these bug reports from our analysis.

4.2.2 Study Design of Qualitative and Quantitative Analysis

Motivation: A typical bug report contains 16 structured and unstructured features [23]. The reporters need to report the features that are suitable for

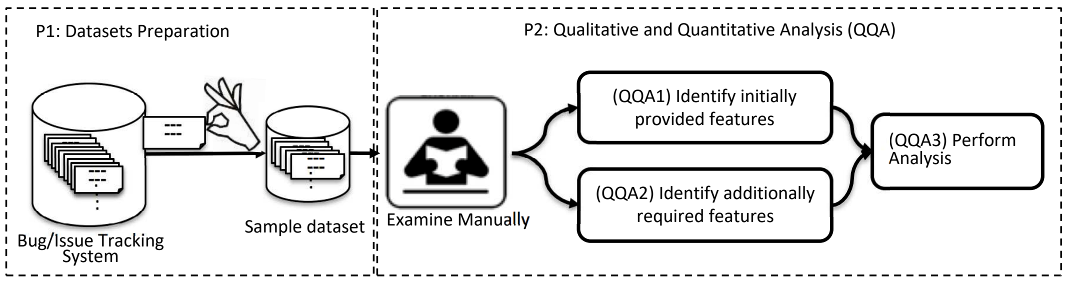

P1: Datasets Preparation
Bug/Issue Tracking System
Sample dataset
P2: Qualitative and Quantitative Analysis (QQA)
Examine Manually
(QQA1) Identify initially provided features
(QQA2) Identify additionally required features
(QQA3) Perform Analysis

Figure 4.1: An overview of our study design

localizing bugs. However, the reporters often omit unstructured features needed by the developers when fixing the bugs. Consequently, the developers need to additionally collect these features, as shown the motivating example in section 2.4. Thus, to understand key features to write a good bug report in this chapter, we conduct a qualitative and quantitative analyses with five OSS projects from the Apache (Camel, Derby, and Wicket projects) and the Mozilla (Firefox, and Thunderbird projects) ecosystems. In particular, we investigate: (1) the features reporters frequently provide in an initial bug report. (2) the features reporters frequently omit, but that developers require after the initial submission; and (3) the impact of the additional required features on bug-fixing process.

Approach: Our qualitative analysis focuses on 10 unstructured features (see Table 4.2) for the sampled bug reports described in Section 3.1. These unstructured features are crucial to developers when fixing bugs [23]. Figure 4.1 provides an overview of our qualitative analysis process (QQA1, QQA2, and QQA3). In QQA1 and QQA2, we identify the initially provided and additional required unstructured features. Then, we divide the bug reports into two groups, namely those without and those with additional required features. Finally, we investigate the differences in bug-fixing time, the number of comments (i.e., comments made by developers and reporters during bug fixing), and the number of commentators (i.e., developers and reporters who participated in discussions during the bug fixing) between the two groups in order to understand the impact of the additional required features on the bug-fixing process.

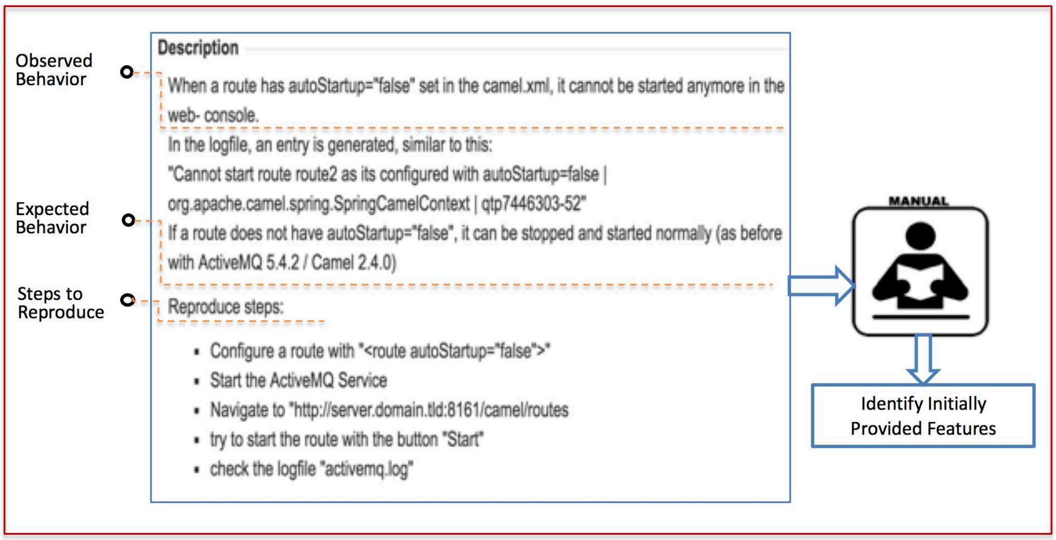
(QQA1) Identify the unstructured features in the initial submission.

Table 4.2: The top 10 most important features of a bug report

Feature	Description
Steps to Reproduce (STR)	A clear set of instructions that the developer can use to reproduce the bug
Stack Trace (ST)	A stack trace produced by the application, most often when the bug reports a crash in the application
Test Case (TC)	One or more test cases that developers can use to determine whether they have fixed the bug
Observed Behavior (OB)	What the user saw happen in the application as a result of the bug
Screenshot (SS)	A screenshot of the application while the bug is occurring
Expected Behavior (EB)	What the user expected to happen, usually contrasted with Observed Behavior
Code Example (CE)	An example of code that can cause the bug
Summary (S)	A short (usually one sentence) summary of the bug
Environment (EN)	The operating system and version the user was using at the time of the error
Error Report (ER)	An error report produced by the application as the bug occurred

We manually identify the unstructured features in the initial bug reports because these features are provided using natural language text in the description. The figure 4.2 shows an example of the description of a bug report. The reporter provided "Observed Behavior", "Expected Behavior", and "Steps to Reproduce" in the description of the bug report. We double-check the identified features for all sampled bug reports to check the correctness of our manual analysis. If there are differences in the identified features for the same bug report, we attempt to reach a consensus on the features.

(QQA2) Identify the additional required features: We manually examine



Description

When a route has autoStartup="false" set in the camel.xml, it cannot be started anymore in the web- console.

In the logfile, an entry is generated, similar to this:
"Cannot start route route2 as its configured with autoStartup=false | org.apache.camel.spring.SpringCamelContext | qtp7446303-52"

If a route does not have autoStartup="false", it can be stopped and started normally (as before with ActiveMQ 5.4.2 / Camel 2.4.0)

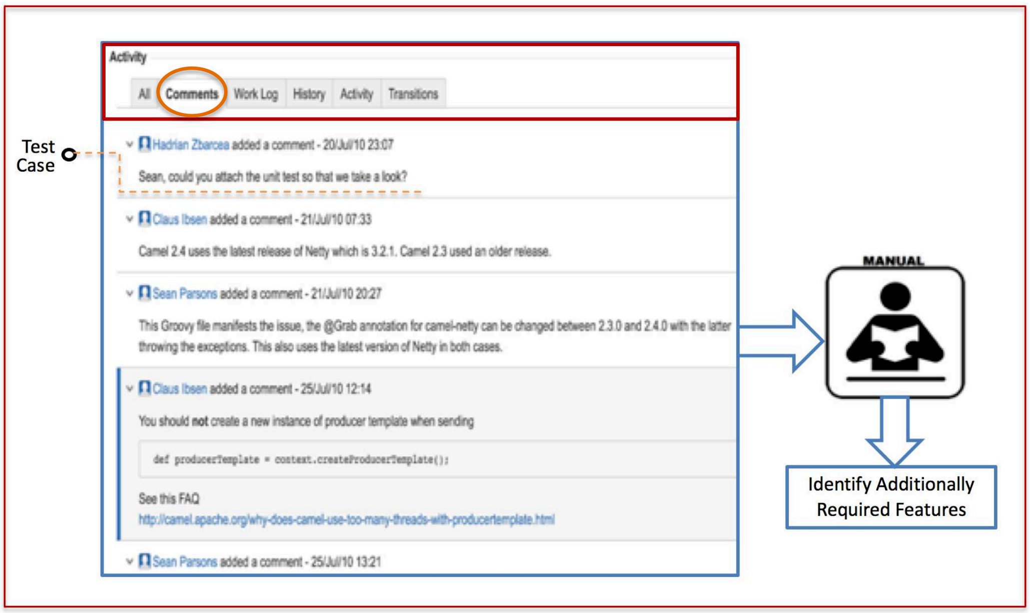
Reproduce steps:

- Configure a route with "<route autoStartup="false">"
- Start the ActiveMQ Service
- Navigate to "http://server.domain.tld:8161/camel/routes
- try to start the route with the button "Start"
- check the logfile "activemq.log"

Figure 4.2: An example of the description of a bug report

bug fixing activities and identify the additional unstructured features that developers requested during bug fixing after the initial submission. The figure 4.3 shows an example of the comments of a bug report. We carefully examine each comment and identify additionally required features. Then, we double-check the identified features and reach a consensus on the features if there are differences for the same bug report.

(QQA3) Perform Analysis To understand the impact of additionally required features on bug fixing process, we divide the bug reports into two groups (i.e., bug reports with additional required features and those without additional required features) for each project. First, we calculate the bug-fixing time for each bug report to understand the difference of bug fixing between these groups. The difference of bug-fixing time between these groups would give us an indication of whether there is any impact of additional required features on the bug fixing process. Then, we calculate the number of comments made by developers during bug fixing and the number of commentators who participated in the bug-fixing process for each group. These two metrics would help us to understand whether the lack of required features in the



Hadrian Zbarcea added a comment - 20/Jul/10 23:07

Sean, could you attach the unit test so that we take a look?

Claus Ibsen added a comment - 21/Jul/10 07:33

Camel 2.4 uses the latest release of Netty which is 3.2.1. Camel 2.3 used an older release.

Sean Parsons added a comment - 21/Jul/10 20:27

This Groovy file manifests the issue, the @Grab annotation for camel-netty can be changed between 2.3.0 and 2.4.0 with the latter throwing the exceptions. This also uses the latest version of Netty in both cases.

Claus Ibsen added a comment - 25/Jul/10 12:14

You should **not** create a new instance of producer template when sending

```
def producerTemplate = context.createProducerTemplate();
```

See this FAQ
http://camel.apache.org/why-does-camel-use-too-many-threads-with-producertemplate.html

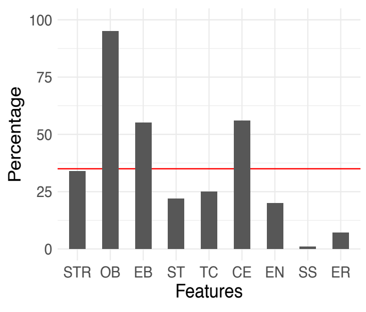
Sean Parsons added a comment - 25/Jul/10 13:21

Figure 4.3: An overview of our study design

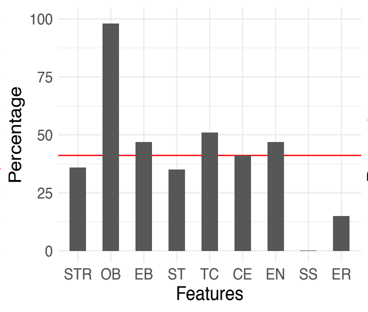
initial bug reports tend to involve more developers in the bug fixing process. Finally, we perform the Mann-Whitney U-test to know the impact of the additional required features during bug fixing. We also calculate Cliff's delta value to know the effect size.

4.3 Analysis Result

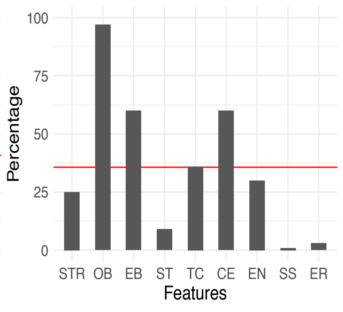
This section presents and discusses the results of our qualitative analysis to identify the features provided in the initial submission of the bug reports, as well as those provided later through discussions between the reporters and the developers during bug fixing. In the qualitative analysis QQA1, we focus on the features that are frequently provided in initial bug reports. Figure 4.4 shows the percentages of each of these features provided in the initial bug reports for the Camel, Derby, Wicket, Firefox, and Thunderbird projects. The red line shows the average percentages of the provided features. They are 35%, 41%, 36%, 44%, and 42% in each project respectively. We found that Observed Behavior frequently



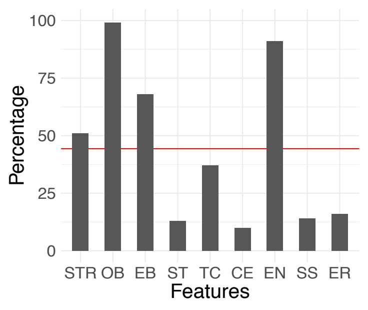
(A) Camel



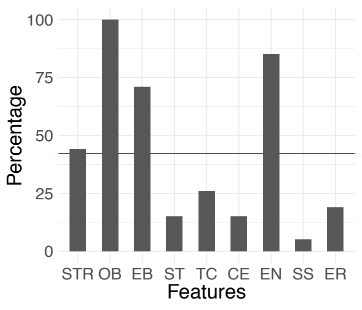
(B) Derby



(C) Wicket



(D) Firefox



(E) Thunderbird

Figure 4.4: The percentages of initially provided features across different projects

exists in the bug reports of the projects. On the other hand, Screenshot and Error Report rarely exist in the bug reports. The rankings of the other features vary across the projects. In these five projects, the most frequently provided features are Observed Behavior, Expected Behavior, Code Example, Steps to Reproduce, Test Case, and Environment. These findings are almost similar to those of existing studies [5, 14].

In the qualitative analysis QQA2, we focus on the additional unstructured features that developers require during bug fixing. Figure 4.5 shows the percentages of these features for each project. The percentages are calculated as follows: $x_i = \frac{\#additional\,requirements\,for\,x_i}{\#bug\,reports\,that\,were\,required\,additional\,features} * 100$. In Figure 4.5, we find that Steps to Reproduce is the additional feature most often required in the Derby, Wicket, and Firefox projects, whereas Test Case is in the Camel and Stack Trace is in Thunderbird projects are most often required features. For all of the projects, Steps to Reproduce and Test Case are the most additional required features

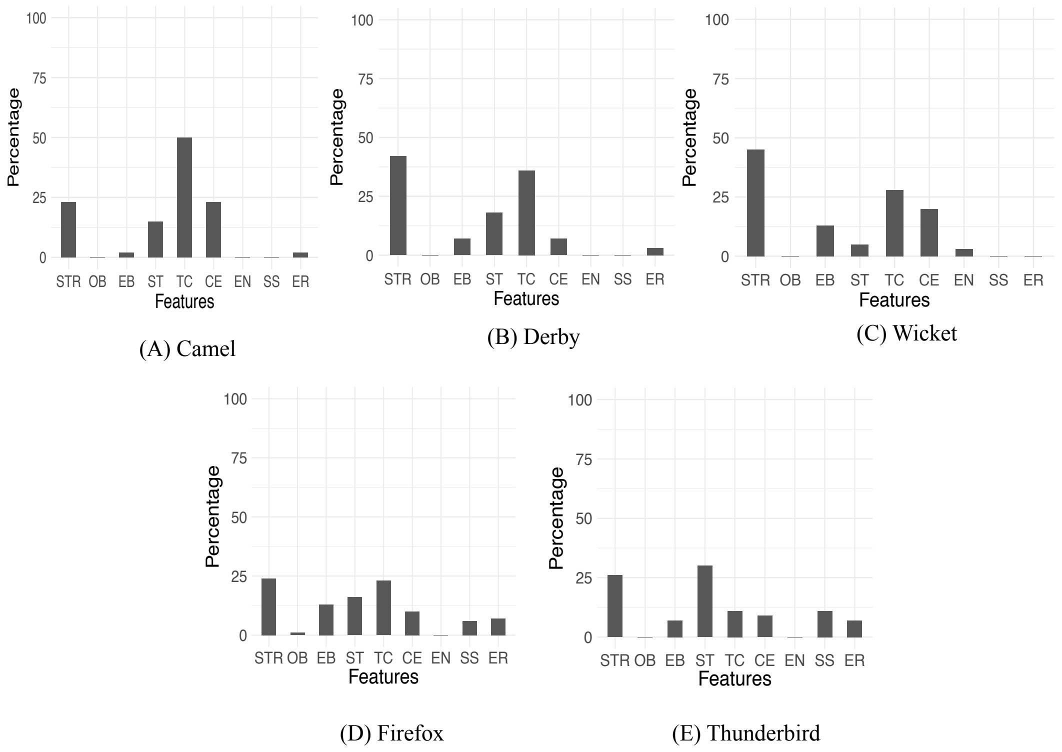
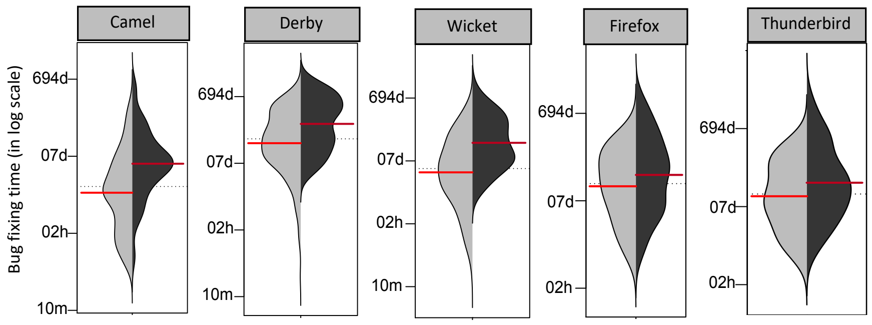


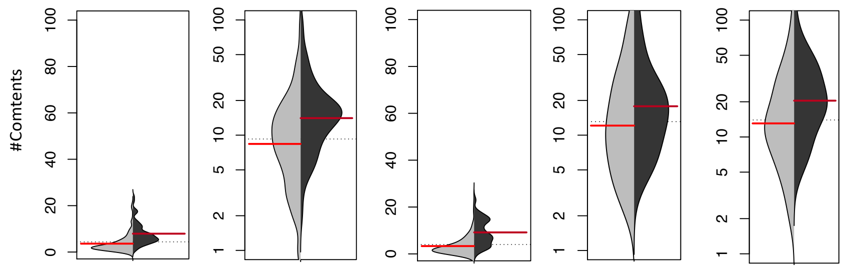
Figure 4.5: The percentages of additional required features during bug fixing across different projects

during bug fixing. We find that Code Example is less often requested in the Derby project compared with other projects. Stack Trace and Expect Behavior are often requested in all five projects. For the other features, we find developers make a very few additional requests during bug-fixing, except Screenshot and Error Report for the Thunderbird project. This suggests that Steps to Reproduce, Test Case, Code Example, Stack Trace, and Expect Behavior are the features requested most often after the initial submission.

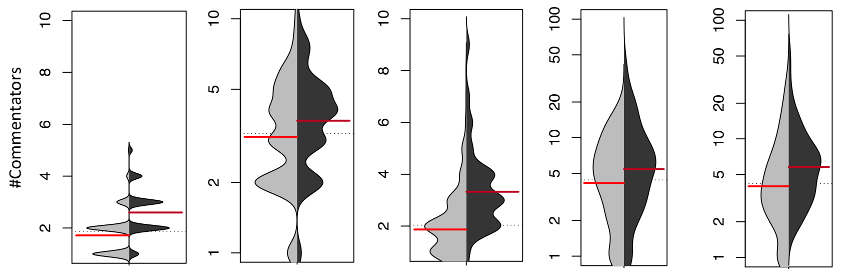
It is obvious that requests for additional features tend to increase bug-fixing time. However, we do not know how the additional features impact the bug-fixing process. To determine this effect, we divide the bug reports into two groups (i.e., bug reports with additional required features and those without additional required features) for each project. Then, we analyze the bug-fixing time, the



(A) The distribution of bug fixing time between bug reports with additional required features (black) and without additional required features (gray) groups



(B) The distribution of number of comments between bug reports with additional required features (black) and without additional required features (gray) groups



(C) The distribution of number of commentators between bug reports with additional required features (black) and without additional required features (gray) groups

Figure 4.6: Impact of additional required features on bug-fixing process

number of comments made by developers during bug fixing, and the number of commentators who participated in the bug-fixing process for each group.

Figure 4.6 (A) shows the distribution of the bug-fixing time using bean plots. The distributions shown in gray and black are those of the bug-fixing times without additional required features and with additional required features, respectively. We find that the median bug-fixing time with the additional required features is much higher than that without additional features across all projects. We observe a significantly different ($p << 0.05$) bug-fixing time between these two groups using the Mann-Whitney U-test with a large (e.g., Derby and Wicket projects) and a medium (e.g., Camel, Firefox, and Thunderbird projects) effect size (see Table 4.3). Figure 4.6(B) shows that the median value of the number of comments (i.e., the number of comments made by developers and reporters during bug fixing) with additional required features is higher than that without additional features. We observe a significant difference in the comment count with a large effect size for the Camel and Wicket projects, a medium effect size for the Derby project, and a small effect size for the Firefox and Thunderbird projects. This indicates that the additional features needed to fix bugs tend to increase the number of comments. In the Figure 4.6 (C), we observe a significant difference in the number of commentators (i.e., the number of developers and reporters who participated in discussions during bug fixing) between the two groups. This indicates that the additional features needed to fix bugs tend to increase the number of developers required to fix the bugs. These findings suggest that the additional required features have a significant impact on the bug-fixing process. Therefore, the features requested most often, such as Steps to Reproduce, Test Case, Code Example, Stack Trace, and Expected Behavior, might be key features in writing a bug report.

Table 4.3: Result of Mann-Whitney U-test and effect size test on with additional features required and without additional features required groups during bug fixing.

Projects	Bug fixing time		Comments during bug fixing		Commentators in bug fixing	
	p-value	Cliff's δ	p-value	Cliff's δ	p-value	Cliff's δ
Camel	p<< 0.05	M	p<< 0.05	L	p<< 0.05	L
Derby	p<< 0.05	L	p<< 0.05	M	p<< 0.05	S
Wicket	p<< 0.05	L	p<< 0.05	L	p<< 0.05	L
Firefox	p<< 0.05	M	p<< 0.05	S	p<< 0.05	M
Thunderbird	p<< 0.05	M	p<< 0.05	S	p<< 0.05	S

Effect size: (L) Cliff's $\delta \geq 0.474$ (M) $0.33 \leq \delta < 0.474$ (S) $0.147 \leq \delta < 0.33$ (N) $\delta < 0.147$

4.4 Discussion

4.4.1 Analysis Result

The purpose of our qualitative analysis is to understand the key features needed to write a good bug report. In QQA1, we identify the frequently provided unstructured features based on initial bug reports. However, we do not yet know whether these features are key features. Reporters might frequently provide unstructured features that are easy to produce. For example, the most frequently reported unstructured feature is Observed Behavior. Sasso et al. [7] found that this feature is easy for reporters to provide. Thus, to obtain a deeper understanding of the key features, we analyze the discussions between the developers and the reporters during bug fixing in QQA2. We find that the reporters often omit five unstructured features (i.e., Steps to Reproduce, Test Case, Code Example, Stack Trace, and Expected Behavior) from their initial submissions. Thus, the developers requested that the reporters provide these features during bug-fixing. We note that Steps to Reproduce and Test Case are requested most often. In reality, Steps to Reproduce is useful because it enables developers to reproduce [15, 23] and understand the bugs. Sometimes, the developers cannot fix a bug without Steps to Reproduce [15]. Test Case is also useful to developers when checking whether the fixed patches are working as expected [1]. A previous survey revealed that 83% and 51%, respectively, of developers consider these two features as helpful when fixing bugs [23]. This suggests that the additional features are particularly important to developers when fixing bugs.

Table 4.4: The Code Example reporting statistics for each project based on the bug reporting experiences of the reporters

Projects	Reporter Type	Code Example (Provided/not provided)	# Bug Reports	Percentages
Camel	Experienced	No	88	43%
		Yes	118	57%
	Non-Experienced	No	63	44%
		Yes	81	56%
Derby	Experienced	No	186	63%
		Yes	109	37%
	Non-Experienced	No	21	38%
		Yes	35	63%
Wicket	Experienced	No	74	38%
		Yes	123	62%
	Non-Experienced	No	51	34%
		Yes	101	66%
Firefox	Experienced	No	131	86%
		Yes	21	14%
	Non-Experienced	No	192	83%
		Yes	14	7%
Thunderbird	Experienced	No	138	81%
		Yes	33	19%
	Non-Experienced	No	123	90%
		Yes	13	10%

Table 4.5: The Code Example reporting statistics for each project based on the bug fixing experiences of the reporters

Projects	Reporter Type	Code Example (Provided/not provided)	# Bug Reports	Percentages
Camel	Contributor	No	60	44%
		Yes	77	56%
	End-user	No	95	46%
		Yes	112	54%
Derby	Contributor	No	132	56%
		Yes	104	44%
	End-user	No	75	40%
		Yes	40	35%
Wicket	Contributor	No	46	48%
		Yes	50	52%
	End-user	No	108	43%
		Yes	145	57%
Firefox	Contributor	No	184	88%
		Yes	26	12%
	End-user	No	139	94%
		Yes	9	6%
Thunderbird	Contributor	No	132	77%
		Yes	39	23%
	End-user	No	129	95%
		Yes	7	5%

To generalize our findings, we set up our case study on the bug reports of different application domains projects from two ecosystems. In our QQA1, we notice that reporters provided Code Example less often in the Mozilla projects compared with the Apache projects. After a closer look, we see that reporters of the selected Apache projects are mostly developers. They sometimes encounter problems during writing codes e.g., bug reports-Wicket-5220 [8] and Wicket-5237 [9]. Thus, they could easily include Code Example in the description of the bug reports. On the other hand, the selected Mozilla projects are client applications and reporters are mostly end-users. They frequently use application and encounter problems e.g., bug reports-bug#216608 [10], bug#220181 [11], and bug#247128 [12]. Thus, they could capture Screenshot and error message (Error Report) and include in the bug reports. However, Code Example is difficult for them to provide in the bug reports [23]. In our QQA2, we see that Screenshot and Error Report requested more often requested in the Mozilla projects compared with the Apache projects. The conversation between the developers (i.e., bug fixers) and the reporters of the bug report-bug#522459 [13] and bug#370401 [14] in Thunderbird project show that these two features help the developers to understand and fix the bugs. Interestingly, Steps to Reproduce and Test Case are the most often additional required features for both ecosystems except the Thunderbird project. We also notice that the top five additional required features slightly vary in the Thunderbird project compared with the other four projects. However, Steps to Reproduce, Test Case, Code Example, Stack Trace, and Expected Behavior are the additional required features requested most in these five projects.

A recent study found the textual difference between the bug reports written by an expert (i.e., anyone who has contributed in the source code of a project) and a non-expert reporter (i.e., anyone who has not contributed in the source code of a project) [69]. Thus, our primary assumption is that the key features such as Code Example, Test Case reporting might also depend on the reporter's

[8]`https://issues.apache.org/jira/browse/WICKET-5220`
[9]`https://issues.apache.org/jira/browse/WICKET-5237`
[10]`https://bugzilla.mozilla.org/show_bug.cgi?id=216608`
[11]`https://bugzilla.mozilla.org/show_bug.cgi?id=220181`
[12]`https://bugzilla.mozilla.org/show_bug.cgi?id=247128`
[13]`https://bugzilla.mozilla.org/show_bug.cgi?id=522459`
[14]`https://bugzilla.mozilla.org/show_bug.cgi?id=370401`

experiences. In order to understand whether the key features reporting depends on the type of bug reporters, we perform a simple qualitative analysis from the two perspectives of bug reporters such as bug reporting experiences and the bug-fixing experiences. To perform our analysis, we classify the reporters of our target sampled bug reports as an experienced reporter if they have submitted at least one bug report before otherwise a non-experienced. Indeed, for each reporter of our target sampled bug reports, we look for his/her past submitted bug reports. If the reporter submitted at least one bug report in the past, we classify the reporter as an experienced reporter. Otherwise, we classify the reporter as a non-experienced reporter. Similarly, we classify the reporters of our target sampled bug reports as a contributor if they have fixed at least one bug report before otherwise an end-user. We take the feature "Code Example" as an example for this analysis because it is one of the difficult features for the reporter [23]. We first investigate whether the bug reporting experiences (experienced vs non-experienced) of bug reporters affects the key features reporting. Table 4.4 shows that the experienced reporter tends to report comparatively higher percentages of Code Example in both ecosystem projects than the non-experienced reporter do. For example, in the case of the Camel project, the experienced reporter provided 46% more Code Example than the non-experienced reporter. In the case of the Thunderbird project, the experienced reporter provided 153% more Code Example that the non-experienced reporter. Then, we investigate whether the bug fixing experiences (contributor vs end-user) of the bug reporters affects the key features reporting. From the Table 4.5, we do not find any relationship, which ascertains that the Code Example reporting depends on the type of bug reporters for the Apache projects. However, we find that the Code Example provided mostly by the contributor in the Mozilla projects. For example, the contributor provided 85% of the total Code Example in the Thunderbird project whereas, the end-user provided only 15%. From this analysis, we see that both bug reporting and bug fixing experiences of bug reporters may affect the key features reporting. Zaman et. al. [25] conducted a case study on security and performance related bugs of the Mozilla Firefox project and observed different characteristics between them. This indicates that the different type of bugs may require different key features to fix. This remains as our future work.

We found that the bug-fixing time with additional required features increases significantly. In reality, the bug-fixing time depends on many factors, such as the complexity of a bug and the developer's expertise. We find that the additional features also affected the bug-fixing time. In the case of the bug report CAMEL-5860 [15], explained in section 2.4, the developer had to wait many days to get the requested feature to reproduce the bug. In another case, when the bug report [16] was assigned, the developer tried to reproduce the bug in different ways using the provided features. However, after failing to do so, the developer had to ask for Steps to Reproduce in order to reproduce the bug. In both cases, the bug-fixing time might have been reduced by providing the feature as part of the initial submission. Zimmerman et. al. [23] also claim that missing features are one of the biggest causes of delays in fixing bugs.

Based on these findings, we conclude that the additional features requested most often, such as Steps to Reproduce, Test Case, Code Example, Stack Trace, and Expected Behavior, might be key features when writing a bug report. Therefore, developing an automated approach to predict and suggest these key features to reporters may improve the bug-fixing process. That leads us to develop an approach to predict such features.

4.4.2 Implications

For Reporters:

Through a case study of three projects (Camel, Derby and Wicket) from the Apache ecosystem and two projects (Firefox and Thunderbird) from the Mozilla ecosystem, we identify five key features that developers often request during bug fixing. Bug reporters can get a good understanding of key features that are important for developers to fix bugs from these findings. Reporters should pay attention to submit the initial bug reports because missing these key feature significantly affect the bug fixing process.

For Researchers:

Recently, IR (Information Retrieval) based automated techniques have gained popularity to select appropriate developers [16], localize bugs [18] and predict bug-

[15]https://issues.apache.org/jira/browse/CAMEL-5860
[16]https://issues.apache.org/jira/browse/CAMEL-1199

fixing effort [19]. However, the performance of these techniques largely depends on the good quality bug reports. Our analysis reveals that the average percentages of provided features in the initial bug reports are 35%, 41%, 36%, 44%, and 42% in Camel, Derby, Wicket, Firefox, and Thunderbird, respectively. Developers were requested for the additional features on average 20% bug reports. Developers are suffering from collecting features during bug fixing. Thus, researchers also need to focus on developing tools/techniques to support reporters to make good bug reports for making the existing tools/techniques effective and actionable.

4.5 Threats to Validity

We now discuss the threats to the validity of our study.

4.5.1 External validity

Threats to external validity relate to the generalizability of our results. We have studied five OSS projects from two ecosystems. Thus, our results may not generalizable to all software systems especially proprietary systems. To combat the potential bias, we first followed strong selection criteria (see Section 3) to select the studied projects from two ecosystems such as Apache and Mozilla. This ensured that our case study included projects from different application domains. Second, we follow specific criteria (see Section 3) to select bug reports from each project in order to design data sets that exclude noisy and biased bug reports. The contents of bug reports might depend on the types of bug reporters (e.g., experienced, non-experienced reporters, contributor, and end-user). Our datasets include a rational proportion of bug reports for each type of bug reporters, which might mitigate such a threat. Third, we use the standard sampling technique with a 95 percent confidence level to calculate the sample size. This ensured a rational subset for each studied project.

4.5.2 Internal validity

Our concerns related to internal threats are the correctness of our manual analysis results and errors. We manually analyzed the bug reports of five projects and identified frequently provided and additional required features for fixing bugs. Like any human activity, the features identification may be prone to human error or bias. To alleviate this threat, we double-check the identified features for all sampled bug reports. If there are differences in the identified features for the same bug report, we attempt to reach a consensus on the features. We also computed Cohen's Kappa value to evaluate the inter-rater agreement. The Cohen's Kappa values are 82%, 80%, 79%, 80%, and 87% for Camel, Derby, Wicket, Firefox, and Thunderbird, respectively, which showed excellent inter-rater agreement. However, there could still be errors that we may not notice.

4.6 Chapter Summary

Our goal of this chapter is to understand the key features that reporters should provide in the description to make a good bug report. To achieve this goal, we first perform a qualitative and quantitative analysis of five OSS projects from two ecosystems to investigate the key features of a bug report by examining bug-fixing activities. Our analysis reveals that Steps to Reproduce, Test Case, Code Example, Stack Trace, and Expected Behavior are the additional required features that reporters most often omit from their initial submissions. Our statistical analysis suggest that the additional required features have a significant impact on the bug-fixing process. Therefore, developing an automated approach to predict and suggest these key features to reporters may improve the bug-fixing process. That leads us to develop an approach to predict (see chapter 6) and recommend key features (see chapter 7) that reporters should provide in the description of the bug reports. Performing qualitative and quantitative analysis on the bug reports of other ecosystems' projects would increase generalization our findings.

5 Investigating Key Features of High Impact Bugs (HIB) Report[1]

The goal of this chapter is to conduct an empirical study to investigate key features according to high impact bugs. To achieve the goal, we manually examine the HIB reports and perform both qualitative and quantitative analysis in the Apache Camel project. Our main findings include: (1) we find four types of features are the most requested information from developers when they fix HIB; (2) the requested additional information significantly influences bug fixing time.

5.1 Introduction

Bug reports are the primary means through which developers triage and fix bugs. Nowadays, software projects are receiving bug reports on a daily basis. In a large and evolving software system, the large number of bug reports typically exceed the available project resources. Accordingly, some bugs might be dealt with a long term delay or not at all [39]. To avoid unnecessary delay to fix bugs, which impact to product and development process, we focus on High Impact Bug (HIB). Existing studies defined six types of HIB and they found that HIB should be fixed quicker than other bugs in software development [40] [41] [1] [42] [43] [44]. For example, a security bug (one of the types of HIB) should be fixed faster than other non-HIB because it allows unauthorized access to the system. Davies et al. found that bug reports are neither complete nor accurate through a case study on four

[1]This chapter based on my master thesis work. We include this work with the dissertation for the readers' better understanding of our research

big scale and successful open source projects (Eclipse, Firefox, Apache HTTP, and Facebook API) [5]. Thus, fixing a HIB sometimes become complicated because of the low-quality bug reports.

Different types of bugs (e.g. Performance and Security bugs) vary from each other [2]. Therefore, our hypothesis is features that developers may find different features useful in different types of high impact bugs. If we can find out the most useful features in each type of high impact bug then we can suggest reporters provide at least the most useful features in the bug reports.

To the best of our knowledge, there is no case study on revealing useful features set for each type of high-impact bugs. Therefore, we motivated to do an empirical case study on high-impact bug reports to improve the content of bug reports. As the first step of our research, we try to reveal useful features for each type of high impact bug by analyzing historical bug reports.

Our intuition is that we may discover insightful information to improve bug report quality by analysis historical bug reports and developers activities during the bug fixing process. In this study, we focus on HIB reports and conduct an empirical study to understand how we can make a good HIB report by providing the minimum number of features. We manually analyzed six types of HIB reports of the Apache Camel project and checked the contents of each bug report. We also analyzed the conversations between developers and reporters to understand the features that have been requested when the developers fix bugs. We address the following three research questions:

RQ1: What are the features that reporters provide in each type of high-impact bug?

The previous research revealed useful 10 features for developers to fix a common bug [3] based on a survey of experienced developers. Another case study found that different types of bugs (e.g. Performance and Security bugs) differ from each other [25]. Our hypothesis is that key features may also differ among HIB to fix.

RQ2: Which features did reporters often miss to submit in each type of high-impact bugs at initial submission?

Reporters do not know which features are useful when fix HIB. Hence, developers often ask for the additional features with the initial bug reports to the reporters. We address this research question to understand what features developers often

request in each type of HIB reports.
RQ3: Does asking additional features influence to bug fixing time?
High-quality bug reports would be crucial to fix HIB in time. Developers' requests for additional features may increase bug-fixing time. We explore that how significantly developers' requested additional features affect bug-fixing time.

Our findings show that Test Cases, Code Examples, Steps to Reproduce, and Stack Traces are the most additionally requested features and the developers' requests significantly increase bug fixing time.

5.2 Background and Definition

In this section, we first briefly introduce high impact bugs. Then, we describe our motivation for this research.

5.2.1 High-impact Bug (HIB)

A bug is considered as a high-impact bug (HIB) if it highly impact on software processes, product, or end-users. Software engineering researchers have introduced different HIB based on their impact [40] [41] [1] [42] [43] [44]. We list the different types of HIB as follows:

1. **Surprise bug:** A surprise bug [41] is a new concept on software bugs . It can disturb the workflow and/or task scheduling of developers, since it appears in unexpected timing (e.g., bugs detected in post-release) and locations (e.g., bugs found in files that are rarely changed in pre-release).

2. **Dormant bug:** A dormant bug [42] is also a new concept on software bugs and 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). Another research fournd that 33% of the reported bugs in Apache Software Foundation (ASF) projects were dormant bugs [34].

3. **Blocker bug:** A blocking bug is a bug that blocks other bugs from being fixed [46]. It often happens if a dependency relationship exists among

software components. Since a blocking bug inhibit developers from fixing other dependent bugs, it has a high impact on developers' task scheduling.

4. **Security bug:** A security bug [70] can raise a serious problem which often impacts on uses of software products directly. It exploits to gain unauthorized access or privileges in the systems. In general, security bugs are supposed to be fixed as soon as possible. Some of the terms and phrases such as "Attack", "Vulnerability", "Hack", "Unauthorized Access", "Security Break", "Crash", "Data loss", "Violate" that bug reporters use to describe security related bugs.

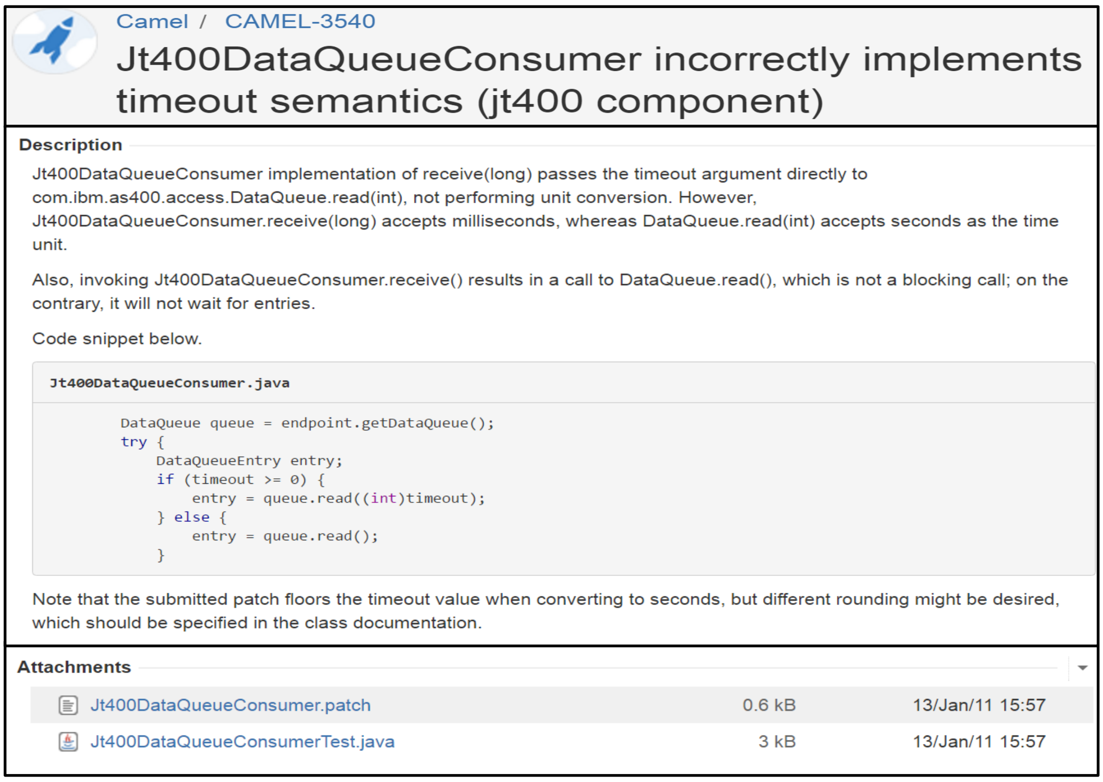
5. **Performance bug:** A performance bug [71] is defined as "programming defects that cause significant performance degradation." The "performance degradation" contains poor user experience, lazy application responsiveness, lower system throughput, and needles waste of computational resources. Some of the terms and phrases such as "Performance", "Slow", "Hang", "Memory leak", "Energy Leak", "Destroying my battery", "Application Not Responding", "Memory Overflow", "Buffer Overflow" that bug reporters use to describe performance related bugs.

6. **Breakage bug:** A breakage bug [41] is "a functional bug which is introduced into a product because the source code is modified to add new features or to fix existing bugs". A breakage bug can cause usable functions in old versions unusable after releasing new versions.

5.2.2 Motivation of this research

Developers depend on the content of bug reports to localize and fix the bugs. Incomplete or Inaccurate bug reports are difficult for developers to understand the bugs. As a result, sometimes, bugs do not get fix, take longer time than expected to fix, fix incorrectly and reopen it later. In some cases, developers make a request to reporters to provide more information. It also increases the bug-fixing time. In case of high impact bugs, these types of unexpected activities are more expensive because high impact bugs have a higher impact than others have. Therefore, bug reports with useful information play a vital role to fix high impact bugs.

For example, the figure 5.1 shows a bug report Camel-3540 that extracted from Apache Camel. It is a performance bug. This bug report is detailed and contains


Camel / CAMEL-3540

Jt400DataQueueConsumer incorrectly implements timeout semantics (jt400 component)

Description

Jt400DataQueueConsumer implementation of receive(long) passes the timeout argument directly to com.ibm.as400.access.DataQueue.read(int), not performing unit conversion. However, Jt400DataQueueConsumer.receive(long) accepts milliseconds, whereas DataQueue.read(int) accepts seconds as the time unit.

Also, invoking Jt400DataQueueConsumer.receive() results in a call to DataQueue.read(), which is not a blocking call; on the contrary, it will not wait for entries.

Code snippet below.

Jt400DataQueueConsumer.java

```
        DataQueue queue = endpoint.getDataQueue();
        try {
            DataQueueEntry entry;
            if (timeout >= 0) {
                entry = queue.read((int)timeout);
            } else {
                entry = queue.read();
            }
```

Note that the submitted patch floors the timeout value when converting to seconds, but different rounding might be desired, which should be specified in the class documentation.

Attachments

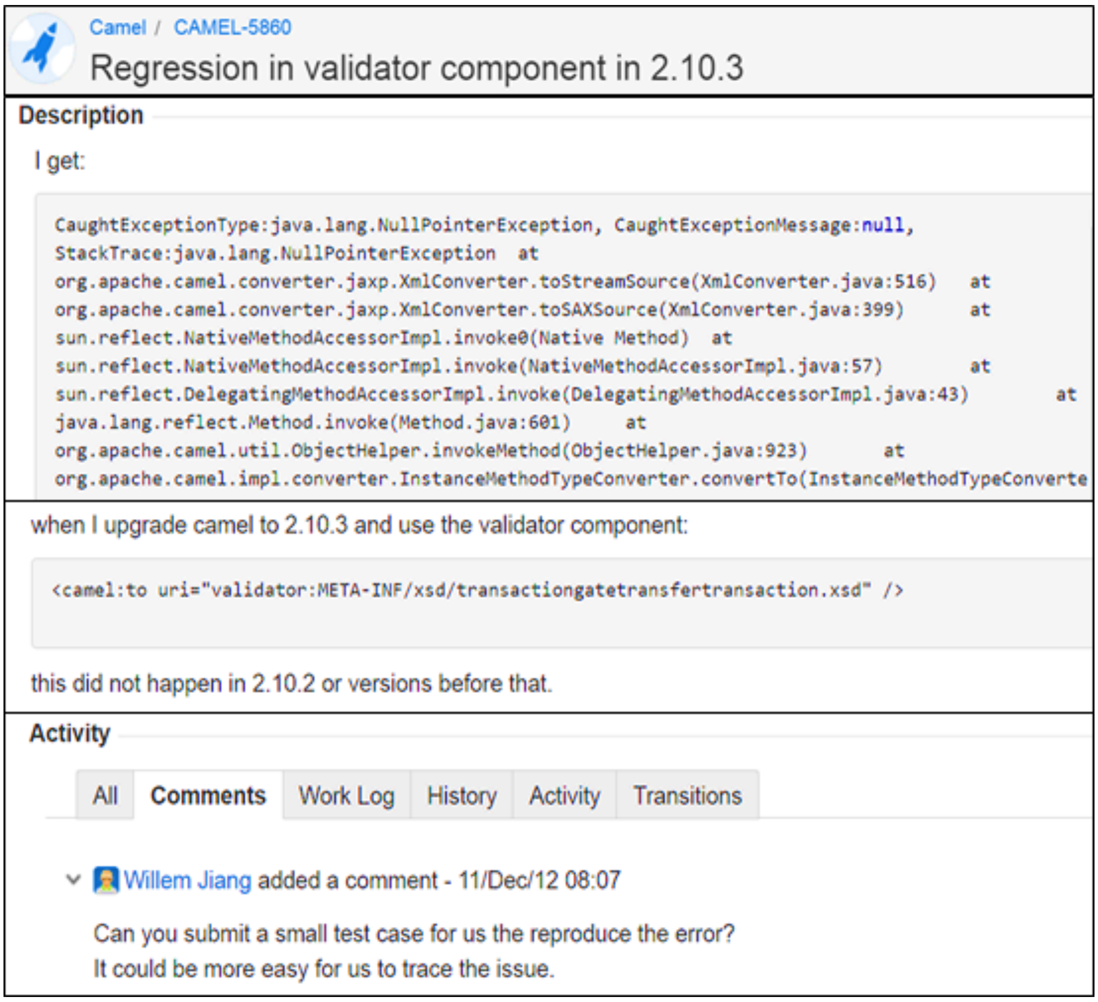
Jt400DataQueueConsumer.patch	0.6 kB	13/Jan/11 15:57
Jt400DataQueueConsumerTest.java	3 kB	13/Jan/11 15:57

Figure 5.1: The description of the bug report Camel-3540 of the Apache Camel project

clear information about the bug. The reporter provided code snippets to describe where the exact problem that happened. The reporter also attached a test case in the description of the report that the developer to test after fixed the bug. This type of bug report is easy to understand for developers. The developer fixed the bug on the same day of reporting.

The figure 5.2 shows an example of bug report Camel-5860 extracted from the Jira of the Apache Camel project. It is a Breakage as well as a Surprise bug.

This bug report was created on December 10, 2012. The reporter provided Stack Traces and very minimal information about the bug. The developer could not understand the exact problem clearly. He also could not understand how


Camel / CAMEL-5860

Regression in validator component in 2.10.3

Description

I get:

```
CaughtExceptionType:java.lang.NullPointerException, CaughtExceptionMessage:null,
StackTrace:java.lang.NullPointerException   at
org.apache.camel.converter.jaxp.XmlConverter.toStreamSource(XmlConverter.java:516)    at
org.apache.camel.converter.jaxp.XmlConverter.toSAXSource(XmlConverter.java:399)       at
sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)  at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)         at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)        at
java.lang.reflect.Method.invoke(Method.java:601)     at
org.apache.camel.util.ObjectHelper.invokeMethod(ObjectHelper.java:923)       at
org.apache.camel.impl.converter.InstanceMethodTypeConverter.convertTo(InstanceMethodTypeConverte
```

when I upgrade camel to 2.10.3 and use the validator component:

```
<camel:to uri="validator:META-INF/xsd/transactiongatetransfertransaction.xsd" />
```

this did not happen in 2.10.2 or versions before that.

Activity

Willem Jiang added a comment - 11/Dec/12 08:07

Can you submit a small test case for us the reproduce the error?
It could be more easy for us to trace the issue.

Figure 5.2: The description of the bug report Camel-5860 of the Apache Camel project

to reproduce the bug. So, one day later, the developer requested to provide additional information *(Can you submit a small test case for us the reproduce the error?)* about the bug. Two months 17 days (on February 27, 2013) later, the reporter responded and provided Steps to Reproduce. Finally, the developer fixed the bug after 8 days. Actually, bug fixing time depends on many factors such as complexities of bug, the severity of bugs, and the developer's experience. However, It also depends on the quality of bug reports.

In the current bug reporting process, there is no intelligence recommendation system to evaluate the content of bug reports and to notify the missing features at the time of submission. Consequently, in many cases, reporters submit less informative bug reports. To overcome the problem, developing intelligence recommendation is crucial to support reporters to let them know the missing features in the bug reports at the time of submission. To develop a recommendation system, first, we need to know the useful features according to high impact bugs to train the system. However, the problem is we do not know which features are useful for which categories of high impact bugs.

The previous research revealed the top 10 features that developers find useful during bug fixing [1]. It is difficult for reporters to provide each of the top 10 features in each bug report in the context of open source projects because most of the reporters are doing a voluntary job. Bug reporting should be as simple as much as possible. Therefore, previous research is not enough to develop a recommendation system.

Therefore, our hypothesis is features that developers find useful may vary among different types of high impact bugs. If we can find out the most useful features in each type of high impact bug then we can suggest reporters provide at least the most useful features in the bug reports. To the best of our knowledge, there is no case study on revealing useful features set for each type of high-impact bug. Therefore, we motivated to do an empirical case study on high impact bug reports to reveal information how bug report should be filed in the BTS and how less informative bug reports affect on bug fixing process by analyzing historical bug reports and developers activities during bug fixing. As a result, reporters can improve the quality of bug report by providing the minimum number of features.

5.3 Case Study Design

This section describes how we conducted the case study. First, we describe target dataset. Then, we describe the analysis procedure to address research question mentioned earlier.

Table 5.1: No.of bug reports in terms of high-impact bugs

High-impact bug	No. of bug reports
Surprise	128
Dormant	69
Blocker	7
Security	14
Performance	51
Breakage	39
Total	308

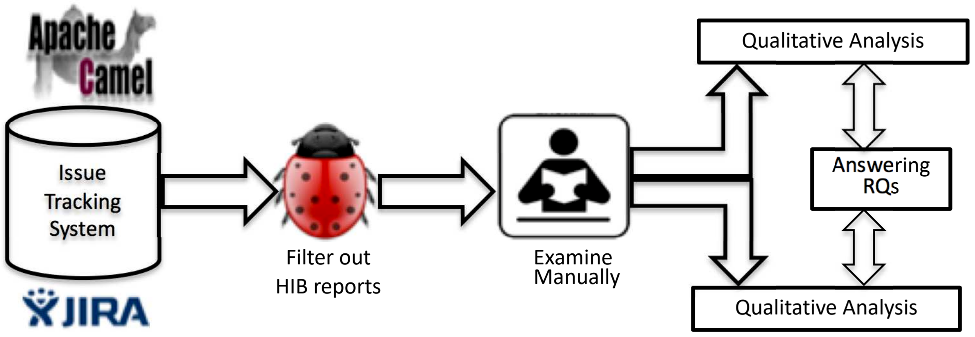


Figure 5.3: An overall analysis procedure of HIB reports

5.3.1 Target Dataset

We conduct a case study on the high impact bug reports of the Apache Camel that is shared by Ohira et al. [44]. The dataset was created by manually reviewing four thousand bug reports for four open source projects (Ambari, Camel, Derby, and Wicket). As kick-off study, we focus on the Camel project to understand key features to report HIB. Table 5.1 shows the statistics of analyzed bug reports. To address our three research questions, we first filtered out HIB reporters from Jira bug tracking system based on [44] for the Apache Camel project. Then, we performed qualitative and quantitative analysis to address our research questions. The figure 5.3 describes overall analysis procedure.

5.3.2 Qualitative Analysis

In our case study, we manually analyzed HIB reports. Typically, a bug report contains a combination of structured (e.g., version, severity, environment, reporter) and unstructured information (e.g., summary, Steps to Reproduce, Observed Behavior). Software engineering researchers have defined this structured and unstructured information as features. Bettenburg et al. [3] surveyed among 156 experienced developers of Apache project, Eclipse project and Mozilla project to examine what features developers expect in a bug report. Based on the feedback from the developers, they revealed the top 10 most important features that developers found useful for bug fixing. Table 2.2 shows the top 10 most important features to fix a bug. Among these 10 features, summary and version are equally important for all kind of HIB reports. So, we excluded these two features and included one feature named Environment (EN) in our study.

However, we believe that all features are not equally important for all kind of HIB report. That means features that developers find useful on bug fixing may vary among different type of HIB. In addition, bug reporters face complexities to provide some features in each HIB report. For instance, bug reporters are not able to provide Stack Traces in each HIB report because it does not always produce especially for the performance bug. Therefore, we need to understand the key features for each type of HIB so that reporters can make good quality bugs by providing the minimum number of features. In order to reveal the key features for each type of HIB report, we have conducted a qualitative analysis. Our qualitative analysis is comprised mainly of two phases.

In the first phase for RQ1, we manually examine each HIB report and extract the reported features from the bug reports. Actually, the structured features are easy to extract automatically. On the other hand, the unstructured 10 features are not so easy to extract because those features are provided with natural language text in the description field. Hence, we conduct a manual examination and observe each reported feature. In detail, we observed what features the developer provided and how they provided and what was the missing features for each HIB report.

In the second phase for RQ2, we mainly examined developers activities on bug fixing. During bug fixing, developers and reporters communicate with each other for various purposes such as clarification, gather missing features, status inquiry.

Usually, when a bug report is assigned to a developer to fix the bug, the developer start work. If the bug report is not well-written and does not contain sufficient information to localize and to fix the bug, developer pause his work and request to provide the missing features. Then, the reporter provides the requested feature. This type of communication kills developer's valuable time. The idea behind analyzing additional request is to understand developer need to fix each type of HIB. Mining frequently developers request habit for additional features can be an important factor to understand key features for each type of HIB.

5.3.3 Quantitative Analysis

During our qualitative analysis for RQ1 and RQ2, we found that sometimes reporters did not provide some features at initial bug report submission. In order to collect the missing features, developers make an additional request to the reporters to provide them. We believe that it affects overall bug fixing. Every bug reporter and project manager assume some level of delay for the bug fix. However, the excessive delay is not tolerable to fix a bug, especially for HIB.

For this reason, we conduct a quantitative analysis to understand whether the request for additional features significantly affects on bug fixing. In detail, we compare the bug fixing time difference between HIB reports with additional feature request (With Request) and HIB reports with no additional feature request (Without Request). The idea behind doing this is to make sense reporters about the aftermath of submitting informative or less informative HIB report. If it significantly affects on bug fixing then reporters will be more careful to file HIB report as well as researchers will be encouraged to do research on how to mitigate the effect.

5.4 Results

RQ1: What are the features that reporters provide in each type of high-impact bug?

To answer RQ1, we carefully examined features in bug reports by reading manually. Figure 5.4 shows how often HIB reports contains each key features. We found average percentages of reported features in Performance, security, Breakage,

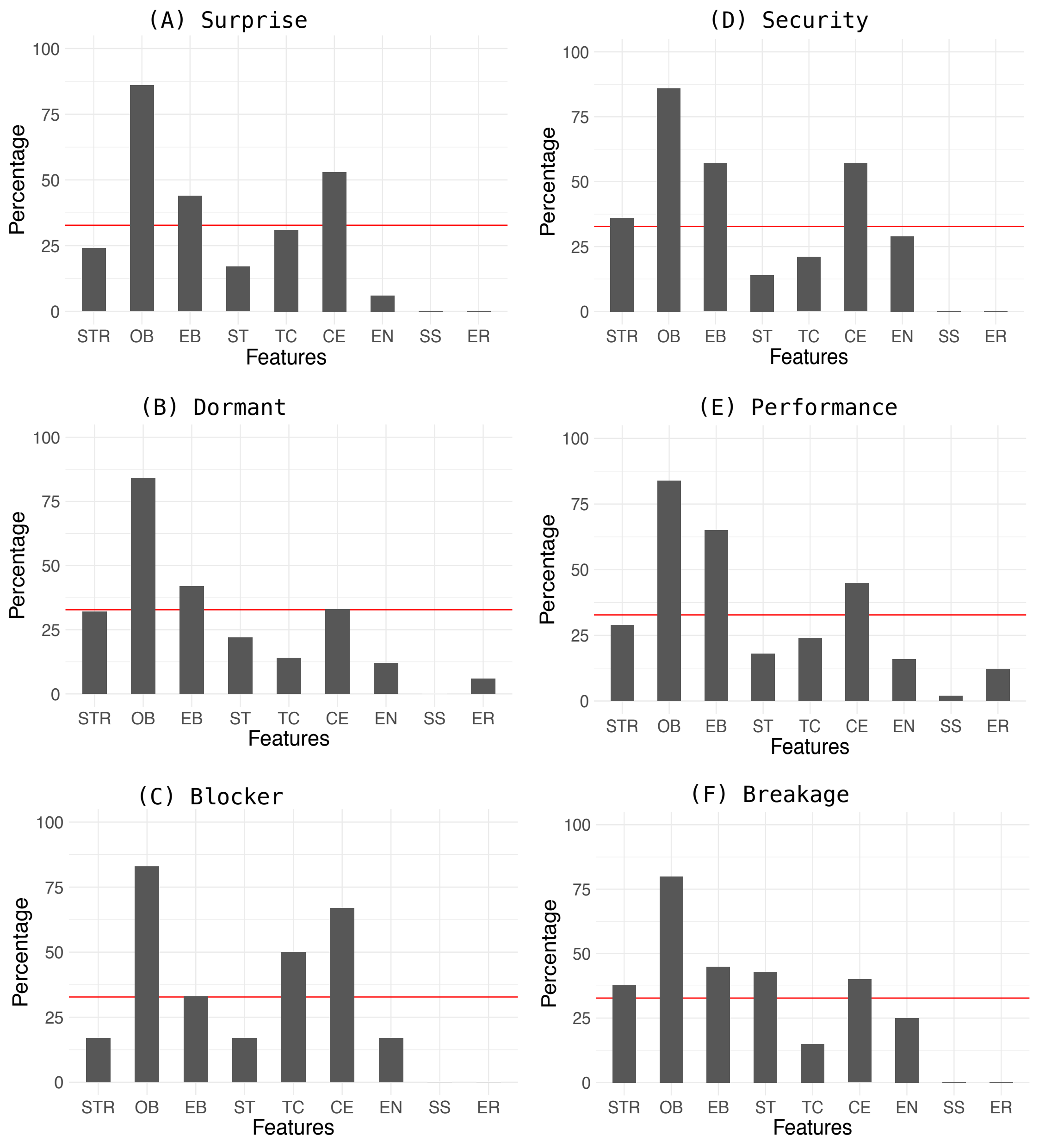


Figure 5.4: Observed reported features in bug reports in terms of HIB

Surprise, Dormant, and Blocker bug reports are 33%, 33%, 32%, 29%, 27%, and 31% respectively (red line). In particular, *Observation behaviour* is the most frequent features in HIB reports. Also, *Expected behaviour* and *Code example* are the next frequent features in HIB reports. On the other hand, *Screenshot* and *Error report* are the least frequent features in HIB reports. The other features are different ranking depends on the type of HIB. For example, while *Test code* is often submitted for only Blocker bug, it is often not submitted for the other type of bug.

RQ2: Which features did reporters often miss to submit in each type of high-impact bugs at initial submission?

To answer RQ2, we examined additional features which developers often asked a reporter to submit. We found that developers made additional request for around 19% HIB reports. Table 5.2 shows the percentage of additionally requested features in each type of HIB. The percentage of additional request for the feature, $x_i = \frac{\#Request\,for\,x_i}{\sum_{i=1}^{i=n}(\#Request\,for\,x_i)} \times 100\%$ where n is the total number of features. We found that developers often asked to *Steps to reproduce*, *Test case* and *Code example* for any types of HIB. For *Steps to reproduce*, 50% security bug report was asked. For *Test case*, approximately 50% of Performance, Breakage, Surprise and Dormant bugs reports were asked. For *Code example*, 50% security bug reports was asked. In our examination, we found that developers did not make any request to provide additional feature for the Blocker bug reports. Blocker bug might be required less features to fix than other HIB.

RQ3: Does asking additional features influence to bug fixing time?

To answer RQ3, we analyze the impact of bug fix due to the additional request for HIB reports. First, we classified HIB reports into two groups; for additional features request (With Request) and no additional features request (Without Request). Then, we measured bug-fixing time which is the bug report created date to fixed date. Figure 8.1 shows the distribution of bug fixing time among the additional requested group and no requested group. We found that the additional requested group is significantly longer bug fixing time than the no requested group ($p << 0.05$).

Table 5.2: Additionally requested features during bug fixing in the high-impact bug reports

Features	High-impact bugs					
	Performance	Security	Breakage	Surprise	Dormant	Blocker
STR	15%	50%	9%	14%	0%	0%
OB	0%	0%	0%	0%	0%	0%
EB	0%	0%	0%	0%	0%	0%
ST	8%	0%	0%	14%	13%	0%
TC	54%	0%	55%	52%	50%	0%
CE	23%	50%	27%	19%	38%	0%
EN	0%	0%	0%	0%	0%	0%
SH	0%	0%	0%	0%	0%	0%
ER	0%	0%	0%	0%	0%	0%

5.5 Discussion

This section discusses some of our major findings from our case study.

In RQ1, we found out the frequently reported features across all types of HIB. Here, we did not consider the features that reporters provided after the initial submission. We have analyzed bug reports of fixed bugs in this case study. So, we may consider the most frequently reported features are more useful to fix HIB. However, at this stage, it is not clear whether they are really useful. This has lead to doing the second analysis.

In RQ2, we carefully examined developers and reporters each activity during bug fixing. Usually, developers request only for those features during bug fixing that are really helpful. Sometimes, they can not work without them. In this point of view, we can consider that additionally requested features are more useful to developers and have a higher impact on bug fixing. We found that developers made an additional request for Test Cases, Code Examples, Steps to Reproduce, and Stack Traces are higher than others. Therefore, we can consider these four are key features for fixing HIB.

In RQ3, we tried to understand whether the request for additional features significantly affects on bug fixing time. Our significant test results suggest that the request for additional features significantly affects on bug fixing time. We

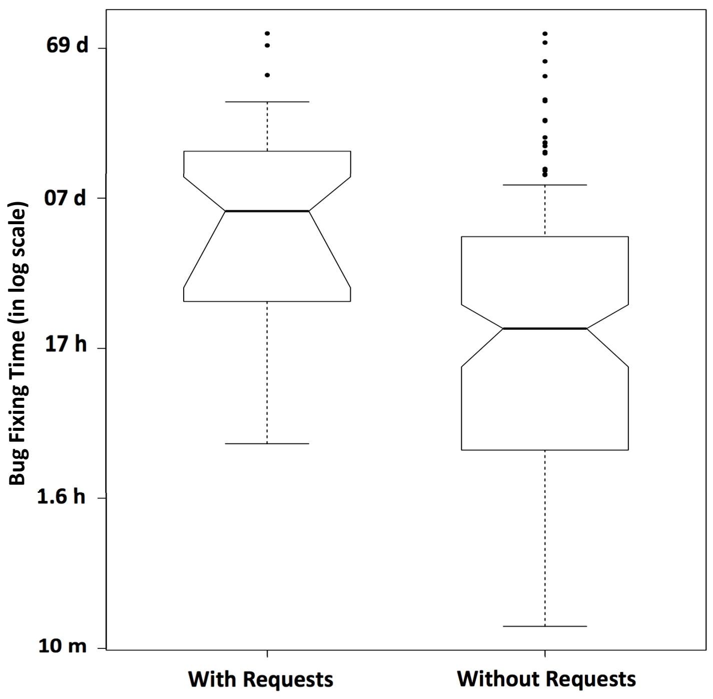


Figure 5.5: Distribution of bug fixing time between additional features request and no-request groups

tried to find out the reasons by analyzing each additionally requested features, requested time, and response time. We found that developers start working to fix the bug and find missing features in the bug reports that are essential to fix the bugs. Then, developers make the request for additional features to reporters and pause the bug fixing activities until the response of reporters. In many cases, reporters take a long time to response the request. This is the reason to affect the request for additional features significantly on bug fixing. Our case study findings will help to reduce the additional request during bug fixing. Findings from the case study will also help to develop automatic tools. If the reporters do not provide key features in the bug report then the tool may generate an automatic suggestion report to notify what additional features should be included in the bug reports. It also helps to write standard guideline on how to fill-up HIB report so that reporters can submit more accurate bug reports in the bug tracking system (BTS).

5.6 Threats to Validity

For our case study we identified the following threats to validity.

5.6.1 External validity

We examined HIB reports of the Apache Camel Project from Jira in our case study. There are some other BTSs such as Bugzilla. Every BTS follows their own convention and style to create and store bug reports. So, our findings may vary for the projects of other BTS.

We conducted a case study on HIB report based on MSR 2015 data showcase paper dataset. The data set contains a limited number of HIB reports for each type. So, our result may not be fully representative of their perspective.

5.6.2 Internal validity

We have analyzed HIB reports of open source projects. Sometimes, proprietary software differs from an open source project. In this regard, our findings from this case study might not be applicable to the proprietary projects. We need to analyze more OSS projects as well as corporate projects to verify the generality of our findings.

5.7 Chapter Summary

In this chapter, we conducted a case study on HIB reports of the Apache Camel project to mine insightful information by analyzing reporters and developers activities. We manually analyzed each HIB report and identified the reported features. Then, we examined developers and reporters each activity during bug fixing. From our investigation, it is clear that in many cases, bug reports are not complete or accurate, and often do not provide features that developers find useful to fix the HIB. We found that for around 19% HIB reports, developers collect useful features by making an additional request that causes significant delay on bug fixing. We also found that Test Cases, Code Examples, Steps to Reproduce, and Stack Traces are the most additionally requested features. Our case study findings suggest that reporters should submit the HIB report more accurately in order to promote the bug-fixing process. Our findings will be helpful to develop automatic tools recommending the bug reporters about the additional features that should be included in the HIB report. It will also help to formulate guidelines for reporters how to fill up bug report form more accurately. Conducting a more comprehensive qualitative and quantitative study on different dimensional projects would increase generalization our case study findings.

6 Predicting Key Features

The goal of this chapter is to build an accurate model to predict whether reporters should provide certain key features in the bug reports by leveraging historical bug fixing knowledge. To achieve the goal, first, we build classification models using four popular text classifications techniques and perform a comparative analysis to reveal the best performing technique in our context based on the prediction performance. In this chapter, we also report the prediction results of our best performing model in the cross-project setting.

6.1 Introduction

The unstructured features are crucial for the developers to triage and fix the bugs [3]. However, reporters often omit these features in their bug reports [4–7]. Thus, developers have to face a number of challeges to triage and fix bugs effectively [4, 13] because developers have to spend much time and effort in order to understand the bugs based on features provided or need to ask reporters to provide additional features [4, 14, 15].

One of the main reasons for the lack of unstructured features in bug reports is inadequate tool support [1, 4, 21, 22]. In order to support reporters, researchers have focused on detecting the presence/absence of unstructured features in bug reports [21, 23, 24]. Zimmerman et al. [23] revealed 10 unstructured features that are important for developers in general. However, there is currently no consensus among software projects and bug reporting systems on the essential mandatory or optional unstructured features that should be part of a bug report [7]. Not all unstructured features are necessarily appropriate for all bug reports [5]. For example, Stack Trace is not generated for all bug reports. Previous empirical studies have found that bug reports contain only between two and six of the top

10 unstructured features [5, 14]. This indicates that not all unstructured features are equally important to fix all bugs. However, selecting those features that should be provided in a bug report is not easy for reporters, especially for novices and end-users. Thus, a tool that only detects the presence/absence of unstructured features is insufficient.

Bug-tracking systems contain historical bug reports, reporters and developer's comments, actions, bug fixing commit records etc [2, 5]. Examining historical bug reports can be a good way to understand which features developers require to fix bugs. However, it is very difficult for reporters to know what features are required based on fixed bug reports from the repository without any automated techniques [21, 23]. This motivates us to develop classification models to predict the key features reporters should provide in initial bug reports based on reports of bugs that have already been fixed.

The summary text has been used successfully to detect similar and duplicate bugs [26, 27]. By examining the contents of bug reports, we can determine which features are required to fix each bug. By applying machine-learning techniques, reporters of new bug reports know which key features need to provide based on the summary text by leveraging historical bug-fixing activities. Thus, in order to help reporters, we build prediction models using Naïve Bayes (NB), Naïve Bayes Multinomial (NBM), K-Nearest Neighbors (KNN), and Support Vector Machine (SVM) text-classification techniques, based on the summary text. Existing studies have found that the performance of prediction models varies between the text-classification techniques [28, 29] depending on the context. Hence, we use the aforementioned four popular text-classification techniques to build and compare prediction models. We evaluate our models using the bug reports of Camel, Derby, Wicket, Firefox, and Thunderbird projects. The main contributions of this chapter are as follows:

1. **We build classification models using popular text-classification techniques to predict key features based on the summary text in bug reports.** We use four popular classifiers to predict whether a reporter should provide certain features based on the summary text in the bug reports. Our models achieve the best f1-scores for Code Example, Test Case, Steps to Reproduce, Stack Trace, and Expected Behavior of 0.70 (Wicket), 0.70

(Derby), 0.70 (Firefox), 0.65 (Firefox), and 0.76 (Firefox), respectively, which are promising.

2. **We conduct a comparative study among different text classification techniques to investigate the best performing text-classification technique to build the key features prediction model.** we identify the best-performing technique of each feature for each project. Thus, for five features, we identify 25 best-performing cases (see column 3 of Table 6.7). Of these, we find that NBM appears in 12 cases as the best-performing technique and 4 cases as no significant difference with the best-performing technique (see the highlighted cells in Table 6.7). Considering all cases, NBM outperforms the other techniques when predicting the key features.

3. **We conduct an experiment to further investigate whether our models can work for predicting the key features in the cross-project setting.** To build prediction models in cross-project setting, we use one projects' dataset for the training the model and another projects' dataset for the testing the performance of the model. Results shows that our best performing model can work successfully for predicting the key features in the cross-project setting.

6.2 Background and Definition

This section, first, describes different types of classification techniques that we use to build prediction models. Then, describes a class imbalance technique that we use to handle class imbalance datasets.

6.2.1 Classification Techniques

We investigate four popular classification algorithms, i.e., naive Bayes (NB), naive Bayes multinominal (NBM), support vector machine (SVM) and K nearest neighbors (KNN).

Naive Bayes (NB): NB is a probabilistic model based on Bayes theorem for conditional probabilities [72, 73]. Naive Bayes assumes that features are independent from one another. Also, all the features are binominal. That is, each feature only has two values of 0 and 1 (in our case, representing whether a word exists in a bug report or not). Based on the above assumptions, given a bug report $BR = (t_1, t_2, ..., t_n)$ (t_i represents a term in the bug report) and a label c_j (in our case: particular key feature present or not), the probability of BR given the label c_j is:

$p(BR/C = c_j) = \prod\limits_{i=1}^{n} p(t_i|C) = c_j)$.

With Bayes theorem, we can compute the probability of a label c_j given BR as follows:

$p(C = c_j|BR) = \frac{p(C=c_j)\times\prod\limits_{i=1}^{n} p(t_i|C)=c_j)}{p(BR}$.

Assuming that the probabilities of different labels and the probabilities of different bug reports are uniform, the above equation can be simplified as:

$p(C = c_j|BR) = \prod\limits_{i=1}^{n} p(t_i|C) = c_j)$.

The probability of word ti given class $c_j(i.e., p(t_i|C = c_j))$ in the above equation can be estimated based on the training data. Next, based on the above equation, we can compute the probability for every label given a new bug report BR, and assign the label with the highest probability to it.

Naive Bayes Multinomial (NBM): NBM is a very similar to NB [72, 73]. However, in NBM the value of each feature is not restricted to 0 or 1, and it can be any non-negative number (in our case, representing the frequency of a word in a bug report). Since NBM can capture more information, it often outperforms NB.

Support Vector Machine(SVM): Given training bug reports, SVM [72, 73] first maps each bug report to a point in a high-dimensional space, in which each feature (in our case: a pre-processed word) represents a dimension. Then, SVM selects the points which have big impact for classification as support vectors. Next, it creates a separating hyperplane as a decision boundary to classify two classes. The separating hyperplane created by SVM has a maximum margin, i.e., it separates the support vectors belonging to the two classes as far as possible. When an unlabeled data instance (in our case: a bug report) needs to be classified, SVM can assign it a label according to the decision boundary.

K-Nearest Neighbors: K-nearest neighbors (KNN) is an instance-based classifier [72, 73]. Its principle is intuitive: similar instances have similar class labels. In our setting, KNN mainly contains three steps. First, similar to SVM, KNN maps all the training bug reports to points in a high-dimensional space. Then, for an unlabeled bug report BR, we find K nearest points to it based on a specific distance metric. In this paper, we use the Euclidean distance as the metric. Euclidean distance between two points is the length of the line segment connecting them. Finally, we determine the label of the key feature of BR by the labels of the majority of its K nearest neighbors.

6.2.2 Imbalance Learning Technique

We notice that our datasets have a class imbalance problem. In order to mitigate the class imbalance problem, we use the following class imbalance learning technique.

SMOTE: SMOTE is a more sophisticated over-sampling method, whose full name is Synthetic Minority Over-sampling Technique [74, 75]. For simplicity, the subsets of data belonging to the minority class and the majority class are denoted by S_{min} and S_{maj}, respectively. Traditional over-sampling methods duplicate data belonging to S_{min}, while SMOTE creates some artificial data which can be assumed to belong to S_{min} based on a specific strategy. Specifically, for each data point x in S_{min}, SMOTE first finds its k-nearest neighbours (data points) belonging to S_{min} and link x with each of these k points to form k line segments (in a multidimensional feature space). Then, SMOTE randomly picks a data point on each line segment. The k new data points can be assumed as belonging to the minority class and be added into S_{min}. Therefore, if there are initially n data points in S_{min}, SMOTE will create $k \times n$ artificial data points and add them to S_{min}. By default, k is set as 5.

6.3 Case Study Design

In this section, we outline our motivation and approach of this study.

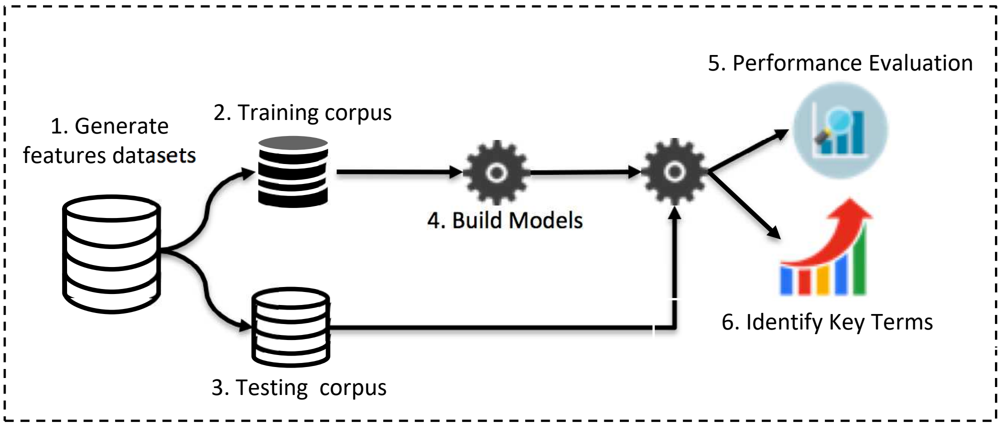


Figure 6.1: An overview of our study design

6.3.1 Motivation of the study:

Developers expect reporters to provide useful unstructured features in their bug reports in order to localize and fix bugs. However, reporters, especially novices and end-users, sometimes find it difficult to do so, because they might not know which features will help developers to fix the bugs [23]. An automated technique that recommeds which features to include in bug reports can reduce the number of missing features [5, 13, 21–23]. Thus, we propose a model that predicts which key features reporters should provide in bug reports that help developers to fix the bugs.

6.3.2 Approach

To predict these key features, we build classification models using four popular text-classification techniques. These models are trained using the "summary," which is one of the unstructured features reporters often use to include key instructions on the detected bugs. Figure 6.1 provides an overview of our study design.

1. **Generate features datasets:** In chapter 4, we manually identify the unstructured features provided in the initial bug reports and comments

section during bug fixing. In this analysis, we build classification models to predict key features that reporters should provide in the description of the initial bug reports. To train our prediction models, we construct a database based on the summaries and the identified unstructured features of the bug reports.

2. **Traning and 3. Testing Corpus** To train and validate our classification models, we use 10-fold cross-validation. First, we split the key features database into 10 equal parts to create a training corpus and a testing corpus. Then, we use nine parts for the training corpus to construct the prediction models in the first round, setting aside one part for the testing corpus. We continue this process until we complete 10 rounds to ensure that each part of the database is used for training and testing corpus. We repeat the whole 10 rounds process of generating training and testing corpus 10 times to ensure the robustness of our approach.

4. **Build Models:** The performance of the prediction models varies between the different text-classification techniques [28, 29, 76] depending on the context. Hence, this study uses four popular text-classification techniques, namely NB [77], NBM [77], KNN [78], and SVM [78] to build models. The reason to choose these techniques is they are classic and diverse algorithms. They represent features in different ways from each other. Existing studies show that they perform well in many text-classification tasks [28, 29, 47, 79]. In addition, the learning strategies of these techniques are different from each other. Although both NB and NBM are based on the Bayes theorem, they represent features in different ways. SVM is a supervised learning model based on the structural risk-minimization principle. Unlike NB or NBM, SVM is a non-probabilistic binary linear classifier. KNN is a distance-based classification algorithm, and differs from NB and NBM. For these reasons, we build prediction models using these techniques and conduct a comparative study to understand which technique performs well in our context. Class imbalance is always a problem in machine learning and can lead to a classifier exhibiting poor performance. Imbalanced learning strategies can be employed to balance an initially imbalanced data set and

help a trained classifier not to be biased to the majority class. Thus, in most cases, they improve the performance of the classifier [50, 80]. There are many imbalanced learning strategies. In our study, we use the synthetic minority over-sampling technique (SMOTE). SMOTE is a more sophisticated over-sampling technique.

5. **Performance Evaluation:** To evaluate the performance of our prediction models, we use traditional evaluation metrics, namely precision, recall, and the f1-score. These metrics are commonly used to evaluate classification performance [72] and can be derived from a confusion matrix. A confusion matrix lists all four possible classification results. If a feature of a bug report predicts correctly, it is a true positive (Tp). If it predicts incorrectly, then it is a false positive (Fp). Similarly, there is a false negative (Fn) and a false positive (Fp) outcome. Based on Tp, Fp, Fn, and Tn, we calculate the precision, recall, and f1-score. Precision is the proportion of correctly predicted features for all bug reports that predicted the feature. Mathematically, precision P is defined as: $P = \frac{Tp}{Tp+Fp}$. Recall is the proportion correctly predicted features in bug reports to the actual number of features in the bug reports. Mathematically, recall R is defined as: $R = \frac{Tp}{Tp+Fn}$. The f1-score is a summary measure that combines the precision and recall. It evaluates whether an increase in precision (recall) outweighs a reduction in recall (precision). Mathematically, the f1-score F is defined as: $F = \frac{2*P*R}{P+R}$.

   Baseline model: Software engineering researchers use random predictors to compare the performance of their prediction models [41, 47, 81]. Since our model is the first to predict key features, we also use a random predictor as a baseline model to evaluate the prediction performance of our model. The precision of a particular feature for random prediction is the percentage of the feature present in the data set. Since the random prediction is a random classifier with two possible outcomes (e.g., feature provided or not provided), its recall is 0.5.

6. **Identify key terms:** To identify the key or best discriminator terms, we extract all the terms from the summary of bug reports and then exclude stop words. After performing pre-processing and stemming terms into

root forms, we found 963, 1024, 931, 825, and 847 unique terms for Camel, Derby, Wicket, Firefox, and Thunderbird projects, respectively. The primary assumption is among these terms some are highly discriminatory for each key feature than others. The most discriminator terms could provide a deep insight into each feature. Recently, a large-scale study [82] conducted on 30 feature selection techniques and apply on 18 datasets. They found that correlation based ranking search technique perform well to select the important features. We also use a correlation based ranking search technique with 10-fold cross-validation to identify the top 10 discriminator terms for each key feature.

6.4 Prediction Result

This section presents and discusses the results of our models in predicting the key features that reporters should provide in their initial bug reports. To predict the key features, we build classification models using four text-classification techniques. Our models are trained based on the summary text of bug reports. Table 6.1 shows the total number of unique terms that extracted from the summary of the bug reports (UT), % of initially provided features (IPF) in the description of bug reports, and % of additionally provided features (APF) in the comment section. We consider IPF+APF as the total required features for fixing a bug. Table 6.2, 6.3, and 6.4 shows the median precision, recall, and f1-score of the key feature prediction for the various text-classification techniques for each project.

Table 6.1: The Structure of training and testing datasets (Total number of unique terms extracted from the summary of the bug reports (UT), % of Initially provided features (IPF), and % of additionally provided features (APF)

	Camel			Derby			Wicket			Firefox			Thunderbird		
Features	UT	IPF	APF	UT	IPF	APF	UT	IPF	APF	UT	IPF	APF	UT	IPF	APF
CE	963	53%	4%	1040	40%	1%	931	60%	2%	825	10%	2%	847	15%	1%
TC	963	22%	7%	1040	46%	6%	931	33%	3%	825	37%	4%	847	26%	2%
STR	963	32%	4%	1040	32%	7%	931	23%	5%	825	51%	5%	847	44%	4%
ST	963	20%	3%	1040	33%	3%	931	9%	1%	825	13%	3%	847	15%	5%
EB	963	54%	1%	1040	46%	1%	931	60%	1%	825	68%	3%	847	71%	1%

Table 6.2: Precision (P), Recall (R), and F1-Score (F1) of different classification techniques and projects for the Code Example and Test Case prediction

	Code Example														
	Camel			Derby			Wicket			Firefox			Thunderbird		
Classifiers	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
NBM	0.67	0.63	**0.65**	0.64	0.67	**0.66**	0.68	0.63	0.65	0.23	0.39	**0.29**	0.31	0.45	**0.37**
KNN	0.65	0.42	0.51	0.48	0.81	0.61	0.66	0.45	0.54	0.15	0.65	0.24	0.30	0.41	0.34
NB	0.61	0.64	0.62	0.63	0.52	0.57	0.70	0.70	**0.70**	0.18	0.74	0.29	0.23	0.65	0.34
SVM	0.64	0.55	0.59	0.58	0.65	0.61	0.66	0.60	0.63	0.28	0.26	0.27	0.33	0.32	0.33
Mean	0.64	0.56	0.59	0.58	0.66	0.61	0.67	0.59	0.63	0.21	0.51	0.27	0.29	0.46	0.34
Baseline	0.57	0.50	0.53	0.41	0.50	0.45	0.62	0.50	0.55	0.12	0.50	0.19	0.16	0.50	0.24
	Test Case														
NBM	0.37	0.45	**0.41**	0.64	0.64	0.64	0.42	0.49	0.45	0.50	0.52	**0.51**	0.43	0.45	0.44
KNN	0.33	0.52	0.41	0.59	0.43	0.50	0.45	0.66	**0.53**	0.49	0.52	0.50	0.41	0.52	**0.47**
NB	0.33	0.42	0.36	0.62	0.80	**0.70**	0.47	0.50	0.48	0.56	0.47	0.50	0.34	0.61	0.44
SVM	0.36	0.39	0.37	0.64	0.57	0.60	0.46	0.52	0.49	0.49	0.52	0.51	0.42	0.45	0.43
Mean	0.35	0.44	0.39	0.62	0.61	0.61	0.45	0.54	0.49	0.51	0.51	0.51	0.40	0.51	0.44
Baseline	0.29	0.50	0.37	0.52	0.50	0.51	0.36	0.50	0.42	0.41	0.50	0.45	0.28	50.00	0.36

Table 6.3: Precision (P), Recall (R), and F1-Score (F1) of different classification techniques and projects for the Steps to Reproduce and Stack Trace prediction

Steps to Reproduce															
	Camel			Derby			Wicket			Firefox			Thunderbird		
Classifiers	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
NBM	0.43	0.47	0.45	0.50	0.51	0.51	0.34	0.42	**0.38**	0.67	0.72	**0.70**	0.62	0.65	**0.64**
KNN	0.39	0.60	**0.47**	0.49	0.69	**0.57**	0.31	0.45	0.36	0.59	0.38	0.47	0.57	0.65	0.61
NB	0.42	0.41	0.41	0.49	0.37	0.41	0.31	0.39	0.35	0.60	0.80	0.68	0.66	0.49	0.56
SVM	0.41	0.49	0.45	0.46	0.52	0.49	0.32	0.36	0.34	0.66	0.58	0.62	0.59	0.66	0.62
Mean	0.41	0.49	0.44	0.49	0.52	0.49	0.32	0.40	0.36	0.63	0.62	0.62	0.61	0.61	0.61
Baseline	0.36	0.50	0.42	0.39	0.50	0.44	0.26	0.50	0.34	0.55	0.50	0.52	0.48	0.50	0.49
Stack Trace															
NBM	0.43	0.54	**0.48**	0.54	0.59	**0.57**	0.19	0.42	0.26	0.46	0.57	0.51	0.47	0.59	0.52
KNN	0.32	0.55	0.41	0.42	0.87	0.57	0.20	0.51	0.29	0.69	0.64	0.65	0.31	0.80	0.44
NB	0.35	0.38	0.36	0.52	0.55	0.54	0.13	0.40	0.19	0.22	0.80	0.34	0.29	0.77	0.42
SVM	0.43	0.49	0.47	0.51	0.65	0.57	0.30	0.29	**0.29**	0.69	0.62	**0.65**	0.62	0.60	**0.61**
Mean	0.38	0.49	0.43	0.50	0.66	0.56	0.20	0.40	0.26	0.51	0.66	0.54	0.42	0.69	0.50
Baseline	0.23	0.50	0.32	0.36	0.50	0.42	0.10	0.50	0.17	0.16	0.50	0.24	0.20	0.50	0.29

Table 6.4: Precision (P), Recall (R), and F1-Score (F1) of different classification techniques and projects for the Expected Behavior prediction

Expected Behavior															
	Camel			Derby			Wicket			Firefox			Thunderbird		
Classifiers	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
NBM	0.62	0.61	0.62	0.54	0.49	0.52	0.64	0.61	0.63	0.77	0.76	**0.76**	0.76	0.74	0.75
KNN	0.54	0.21	0.31	0.50	0.62	0.55	0.70	0.16	0.26	0.76	0.51	0.61	0.79	0.35	0.48
NB	0.59	0.82	**0.68**	0.55	0.35	0.43	0.64	0.64	**0.64**	0.74	0.68	0.71	0.76	0.76	**0.76**
SVM	0.63	0.54	0.58	0.53	0.59	**0.56**	0.67	0.55	0.60	0.75	0.70	0.72	0.77	0.74	0.76
Mean	0.59	0.55	0.55	0.53	0.51	0.51	0.66	0.49	0.53	0.76	0.66	0.70	0.77	0.65	0.69
Baseline	0.55	0.50	0.52	0.47	0.50	0.48	0.61	0.50	0.55	0.68	0.50	0.58	0.71	0.50	0.59

The f1-scores shown in bold represent the best f1-score among the various techniques for each project and each feature. In the case of Code Example, our models achieve the best f1-score of 0.70 with NB for the Wicket project. Comparing this result with the baseline model (0.55), we observe that our prediction model provides a 27% improvement. The best f1-scores for the Camel, Derby, Firefox, and Thunderbird projects are 0.65, 0.66, 0.29, and 0.37 respectively, with NBM, whereas the f1-scores of the baseline models are 0.53, 0.45, 0.19, and 0.24 respectively. Thus, our models achieve a 23–51% improvement over the baseline models for predicting Code Example. In the case of the Test Case prediction, our models achieve the best f1-score of 0.41 and 0.51 with NBM for the Camel and Firefox projects, 0.70 with NB for the Derby project, and 0.53 and 0.47 with KNN for the Wicket and Thunderbird projects. Comparing these results with those of the baseline models (0.37, 0.45, 0.51,0.42, and 0.36), we observe that our models achieve a 12–36% improvement over the baseline models. Similarly, our models achieve a 12–33% improvement for Steps to Reproduce, a 36–115% improvement for Stack Trace, and a 16–33% improvement in Expected Behavior over the baseline models in terms of the f1-score. Thus, the improvement of our models over the baseline models is substantial in terms of the f1-score.

Table 6.5: The Accuracy (ACC) and Area under the receiver operator characteristic curve (AUC) of different classification techniques and projects for the Code Example, Test Case, and Steps to Reproduce prediction

Code Example										
	Camel		Derby		Wicket		Firefox		Thunderbird	
Classifiers	ACC	AUC	ACC	AUC	ACC	AUC	ACC	AUC	ACC	AUC
NBM	0.68	0.68	0.78	0.85	0.64	0.66	0.80	0.75	0.77	0.80
KNN	0.57	0.61	0.61	0.66	0.56	0.63	0.55	0.71	0.78	0.71
NB	0.60	0.56	0.74	0.74	0.67	0.63	0.61	0.58	0.61	0.64
SVM	0.60	0.57	0.72	0.70	0.61	0.56	0.86	0.56	0.81	0.58
Mean	0.61	0.60	0.71	0.74	0.62	0.62	0.71	0.65	0.74	0.69
Baseline	0.51	0.50	0.52	0.50	0.53	0.50	0.79	0.50	0.73	0.50
Test Case										
NBM	0.66	0.69	0.65	0.70	0.64	0.68	0.68	0.72	0.76	0.73
KNN	0.56	0.62	0.59	0.63	0.65	0.70	0.66	0.69	0.74	0.74
NB	0.61	0.62	0.66	0.65	0.67	0.66	0.72	0.61	0.64	0.63
SVM	0.62	0.61	0.65	0.64	0.69	0.66	0.67	0.65	0.76	0.65
Mean	0.61	0.63	0.64	0.65	0.66	0.68	0.68	0.67	0.73	0.69
Baseline	0.59	0.50	0.50	0.50	0.54	0.50	0.52	0.50	0.60	0.50
Steps to Reproduce										
NBM	0.63	0.64	0.64	0.68	0.64	0.62	0.74	0.78	0.73	0.78
KNN	0.56	0.61	0.64	0.68	0.62	0.63	0.60	0.72	0.68	0.73
NB	0.66	0.62	0.65	0.61	0.61	0.52	0.68	0.69	0.72	0.69
SVM	0.61	0.59	0.57	0.55	0.68	0.59	0.70	0.68	0.70	0.70
Mean	0.62	0.61	0.62	0.63	0.64	0.59	0.68	0.72	0.71	0.73
Baseline	0.54	0.50	0.52	0.50	0.59	0.50	0.51	0.50	0.50	0.50

Table 6.6: The Accuracy (ACC) and Area under the receiver operator characteristic curve (AUC) of different classification techniques and projects for the Stack Trace and Expected Behavior prediction

Stack Trace										
	Camel		Derby		Wicket		Firefox		Thunderbird	
Classifiers	ACC	AUC	ACC	AUC	ACC	AUC	ACC	AUC	ACC	AUC
NBM	0.74	0.79	0.74	0.80	0.80	0.82	0.86	0.84	0.82	0.86
KNN	0.63	0.68	0.56	0.66	0.74	0.73	0.93	0.91	0.64	0.76
NB	0.68	0.66	0.72	0.70	0.70	0.59	0.56	0.68	0.62	0.59
SVM	0.77	0.75	0.71	0.70	0.87	0.69	0.93	0.84	0.88	0.84
Mean	0.70	0.72	0.68	0.72	0.78	0.71	0.82	0.81	0.74	0.76
Baseline	0.65	0.50	0.54	0.50	0.82	0.50	0.73	0.50	0.68	0.50
Expected Behavior										
NBM	0.64	0.69	0.63	0.67	0.65	0.65	0.76	0.77	0.73	0.71
KNN	0.54	0.59	0.56	0.60	0.52	0.64	0.63	0.63	0.55	0.67
NB	0.63	0.70	0.60	0.59	0.61	0.57	0.70	0.60	0.74	0.63
SVM	0.68	0.71	0.64	0.62	0.63	0.61	0.72	0.70	0.74	0.73
Mean	0.62	0.67	0.61	0.62	0.60	0.61	0.70	0.67	0.69	0.68
Baseline	0.51	0.50	0.50	0.50	0.52	0.50	0.56	0.50	0.59	0.50

Our best performing model achieves the lowest precision of 0.30 with SVM to predict the Stack Trace for the Wicket project. Comparing this result with the baseline model (0.10), we observe that our prediction model provides a 200% improvement. Similarly, NBM produces a 93% better precision than the baseline model for the Stack Trace prediction for the Camel project. Thus, the improvement of our models over the baseline models is substantial in terms of precision. In some cases, we find that the recall values of our best predictor are lower than the values of the baseline models. In practice, there is a trade-off between precision and recall. We can increase precision by sacrificing recall. The trade-off causes difficulties when comparing the performance of prediction models using precision or recall alone [72]. Thus, the f1-score, which is a trade-off between precision and recall, is used as the main metric to evaluate the performance of our prediction model and a random prediction. In very few cases, the f1-scores of our models are lower than those of the baseline models. For example, our model achieves a lower f1-score with KNN when predicting Expected Behavior for the Wicket project than that of the baseline model. This is because of low recall. However, in most cases, our prediction models achieve a much better f1-score compared with those of the baseline models. The f1-scores of our best-performing models range from 0.29 to 0.70 for Code Example, 0.41 to 0.70 for Test Case, 0.38 to 0.70 for Steps to Reproduce, 0.29 to 0.65 for Stack Trace, and 0.56 to 0.76 for Expected Behavior across the projects.

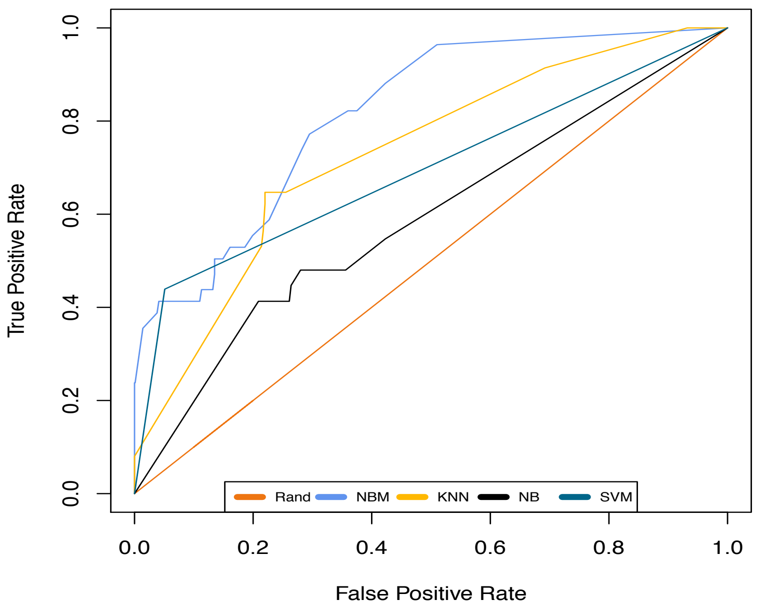
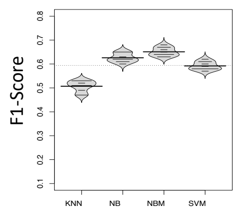
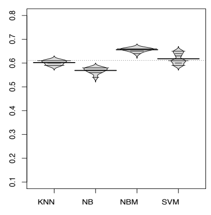


Figure 6.2: The Area under the receiver operator characteristic curve (ROC) for Stack Trace of Wicket project

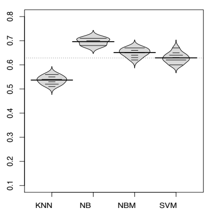
We notice that in some cases, our models achieve low f1-score to predict the key features. To get a better understanding of the prediction performance of our models, we calculate the prediction accuracy (ACC) and Area under the receiver operator characteristic curve (AUC). Table 6.5 and 6.6 shows that our models achieve promising ACC and AUC values of different classification techniques to predict the key features for the different projects. For example, in the case of Code Example for the Derby project, our best model (NBM) achieves 78% accuracy score, which is 51% improvement over the baseline model. The AUC value is 0.85, which is 70% improvement over the baseline model. Table 6.2, 6.3, and 6.4 shows that our models achieve the lowest f1-score for the Stack Trace of Wicket project. However, in this case, our models achieve better ACC and AUC values. The ROC curve in Figure 6.2 also show that our models can predict the key features more accurately than the baseline model.



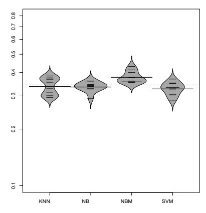
(A) Camel (CE)



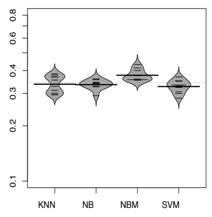
(B) Derby (CE)



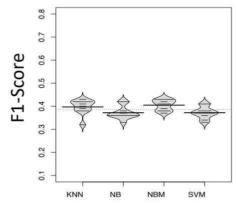
(C) Wicket (CE)



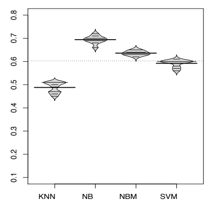
(D) Firefox (CE)



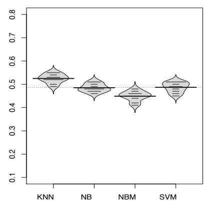
(E) Thunderbird (CE)



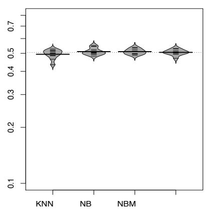
(F) Camel (TC)



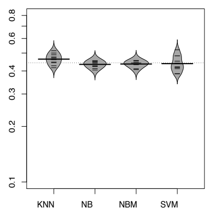
(G) Derby (TC)



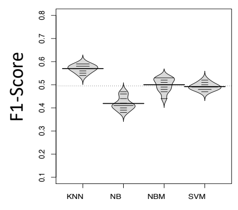
(G) Wicket (TC)



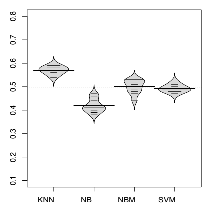
(I) Firefox (TC)



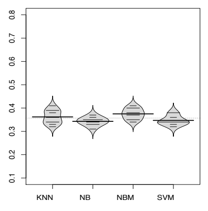
(J) Thunderbird (TC)



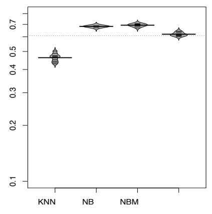
(K) Camel (STR)



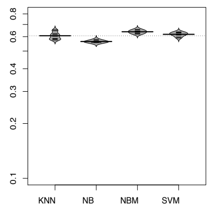
(L) Derby (STR)



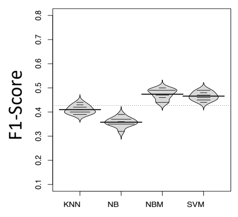
(M) Wicket (STR)



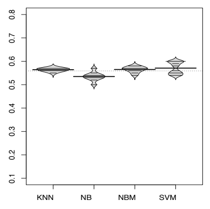
(N) Firefox (STR)



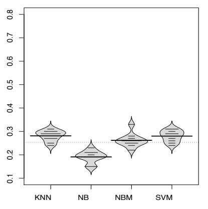
(O) Thunderbird (STR)



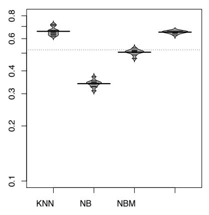
(P) Camel (ST)



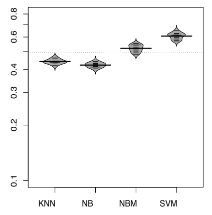
(Q) Derby (ST)



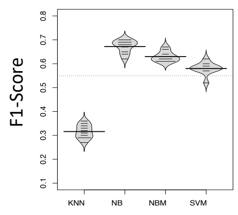
(R) Wicket (ST)



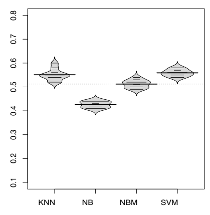
(S) Firefox (ST)



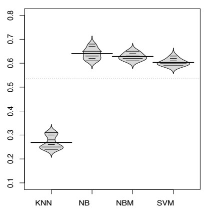
(T) Thunderbird (ST)



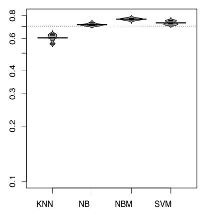
(U) Camel (EB)



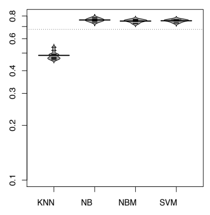
(V) Derby (EB)



(W) Wicket (EB)



(X) Firefox (EB)



(Y) Thunderbird (EB)

Figure 6.3: The difference in performance (f1-score) for predicting the key features among the different text-mining techniques across the projects

Table 6.7: Significant test result between the best classifier and other classifiers for each key feature

.

Projects	Features	Best Classifier	Other Classifiers				
			NBM	KNN	NB	SVM	Baseline
	CE	**NBM**	–	***	*	***	***
	TC	**NBM**	–	–	**	**	***
Camel	STR	KNN	–	–	***	*	***
	ST	**NBM**	–	***	***	–	***
	EB	NB	***	***	–	***	***
	CE	**NBM**	–	***	***	***	***
	TC	NB	***	***	–	***	***
Derby	STR	KNN	***	–	***	***	***
	ST	**NBM**	–	–	**	–	***
	EB	SVM	***	–	***	–	***
	CE	NB	***	***	–	***	***
	TC	KNN	***	–	***	***	***
Wicket	STR	**NBM**	–	–	**	*	***
	ST	SVM	–	–	***	–	***
	EB	NB	–	***	–	**	***
	CE	**NBM**	–	***	–	–	***
	TC	**NBM**	–	–	–	–	***
Firefox	STR	**NBM**	–	***	*	***	***
	ST	KNN	***	–	***	–	***
	EB	**NBM**	–	***	***	***	***
	CE	**NBM**	–	*	***	***	***
	TC	KNN	*	*	–	–	***
Thunderbird	STR	**NBM**	–	*	***	**	***
	ST	SVM	***	***	***	–	***
	EB	NB	–	***	–	–	***

Statistical significance: *p<0.05, **p<0.01, ***p<0.001

In order to investigate the best performing text-classification technique to build the key features prediction model, we conduct a comparative analysis of the four text-classification techniques. Figure 6.3 shows the distribution of the f1-scores among the different text-classification techniques across the projects. We see that our best-performing classifiers significantly outperform over the baseline classifier. However, there is no single best-performing classification technique for predicting key features. For example, in the case of Code Example, NBM performs better than other techniques for the Camel and Derby projects. NB performs better than the other techniques for the Wicket project, but the f1-score of NBM is very close to the f1-score of NB. In order to determine whether the difference is statistically significant, we conduct a statistical significance test between the different classification techniques. To perform this test, we first select the best-performing technique for each feature of each project. Then, we calculate the Mann–Whitney U–test (also called Wilcoxon rank sum test) result for the best-performing technique compared with the others, one by one, to observe whether the p-value is less than 0.05. Table 6.7 shows the significant test results between the best classifier and other techniques for each key feature. We find that in some cases there are no best performing techniques for identifying key features. For example, there is no significant difference in performance between NBM and SVM for the Camel and Derby projects to identify Stack Trace. This indicates that we can choose either technique to build the prediction model. In some cases, we find that the best-performing technique (e.g., NB for Test Case for the Derby project) significantly outperform the other techniques. In these cases, the best-performing technique is highly recommended for building the prediction model. For each feature, we identify the best-performing technique for each project. Thus, for five features, we identify 25 best-performing cases (see column 3 of Table 6.7). Of these, we find that NBM appears in 12 cases as the best-performing technique and 4 cases as no significant difference with the best-performing technique (see the highlighted cells in Table 6.7). Considering all cases, NBM outperforms the other techniques when predicting the key features.

6.5 Discussion

6.5.1 Prediction Result

One of the purposes of our quantitative analysis is to build an accurate prediction model to predict the key features based on the summary of the bug reports. Table 6.2, 6.3, and 6.4 show that the performance of the prediction models varies across the projects. For example, in the case of the Test Case prediction, our best-performing model achieves f1-score of 0.41 for the Camel and f1-score of 0.70 for the Derby project. One of the reasons can be the terms present in the description of the bug reports. To obtain deeper insight, we identify the top-10 discriminative terms for each key feature based on the correlation value. We can see from Tables 6.8, 6.9, 6.10, 6.11, and 6.12 that the different application domain projects share the set of discriminative terms for each key feature. The correlation value of a particular term determines how much the summary of the bug report correlates with the feature. We see that the term "failure" appears in both the Camel and the Derby projects for the Test Case. We further perform a simple qualitative analysis to investigate each of the top-10 terms and examine how many bug reports (i.e., the summaries of the bug reports) contain each of the top-10 discriminative terms in the training documents. We find that the term "failure" appears in 99 bug reports, or 28% of all bug reports for the Derby project. On average, 30 bug reports (the average is calculated as $\frac{1}{10}\sum_{i=1}^{i=10} x_i$, where x_i is the number of summaries that contain the i^{th} term in the training documents) contain the top-10 discriminative terms in the Derby project. On the other hand, the term "failure" appears in only 24 bug reports, or 7% of all bug reports for the Camel project. On average, 13 bug reports contain the top-10 discriminative terms. We find a large difference between the average number of top-10 terms contained in the Derby project and those in the Camel project. Closer inspection reveals that the Derby project contains a large percentage of bug reports that are related to testing failures. Thus, reporters included the terms "failure" (99 times) and "test"(95 times) more often. This might be why the Derby project contains more of the top-10 discriminative terms, on average than the Camel project does. We see in table 6.2 that the prediction performance of Test Case in terms of precision, recall, and f1-score is much better for the Derby project than for the Camel project. This indicates that the terms contained in the summaries of the bug reports might affect the performance of our prediction models.

In the case of Stack Trace, we find that the term “nullpointer” appears in 8%, 14%, and 5% of the summaries for the Camel, Derby, and Wicket projects, respectively. On average, 11, 29, and 8 bug reports contain the top-10 discriminative terms in the summaries for the Camel, Derby, and Wicket projects, respectively. We find that our models achieve comparatively lower precision, recall, and f1-scores when predicting Stack Trace for the Wicket project than for the Derby project. We also find that a smaller percentage of bug reports share the top-10 discriminative terms in the Wicket project than in the Derby project. This indicates that the presence of the discriminative terms in the summary of the bug reports affect the performance of our prediction models.

Table 6.8: Top-10 discriminative terms based on the correlation value of each key feature for the Camel project

Projects	Features	Top-10 discriminative terms
Camel	CE	camel (0.175), property (0.128), sftp (0.123), package (0.119), concurrentmodification (0.119), route (0.117), org (0.111), camelcontext (0.104), specific (0.104), configure (0.104)
	TC	window (0.152), failure (0.140), simple (0.131), problem (0.131), reference (0.131), raise (0.131), main (0.131), content (0.119), bean (0.115), procedure (0.109)
	STR	bean (0.138), ftp (0.121), bug (0.117), run (0.117), read (0.117), issue (0.114), stream (0.114), fill (0.113), url (0.110), endpoint (0.105)
	ST	nullpointer (0.271), nullpointerexcept (0.179), configure (0.162), concurrentmodification (0.162), defaultcamelcontext (0.162), osg (0.153), provider (0.132), regression (0.132), version (0.132), header (0.113)
	EB	java (0.135), pipeline (0.120), attribute (0.120), throw (0.109), minimize (0.109), incorrect (0.108), content (0.108), example (0.104), run (0.104), match (0.104)

Table 6.9: Top-10 discriminative terms based on the correlation value of each key feature for the Derby project

Projects	Features	Top-10 discriminative terms
Derby	CE	column (0.255), test (0.214), serial (0.192), failure (0.186), order (0.178), insert (0.172), clause (0.166), result (0.156), cause (0.155), apache (0.147)
	TC	language (0.167), make (0.164), failure (0.163), expect (0.141), correct (0.138), remove (0.138), test (0.133), found (0.133), drop (0.124), cause (0.116)
	STR	nullpointer (0.160), select (0.141), insert (0.136), ignore (0.135), sql (0.121), distinct (0.121), expression (0.121), unexpect (0.117), derbynet (0.117), timestamp (0.107)
	ST	nullpointerexcept (0.229), incorrect (0.196), test (0.194), assertionfailederror (0.192), nullpointer (0.185), fail (0.178), ibm (0.163), wait (0.159), read (0.151), weme (0.138)
	EB	incorrect (0.180), nullpointerexcept (0.166), sql (0.146), server (0.133), derbytest (0.132), functiontest (0.132), include (0.124), privilege (0.124), java (0.111), example (0.111)

Table 6.10: Top-10 discriminative terms based on the correlation value of each key feature for the Wicket project

Projects	Features	Top-10 discriminative terms
Wicket	CE	encode (0.134), object (0.130), tomcat (0.130), service (0.118), issue (0.118), filterpath (0.106), zip (0.106), double (0.106), log (0.105), catch (0.103)
	TC	wickettester (0.199), path (0.155), render (0.137), component (0.135), model (0.132), enclosure (0.122), datepicker (0.122), setenable (0.118), cookie (0.118), label (0.118)
	STR	modalwindow (0.218), before (0.164), pagelink (0.147), order (0.142), contribute (0.142), redirect (0.130), datepicker (0.130), call (0.121), iframe (0.116), clear (0.116)
	ST	debug (0.227), safari (0.227), nullpointer (0.224), service (0.201), child (0.176), java (0.176), tomcat (0.176), regression (0.161), reguestlogger (0.161), interrupt (0.161)
	EB	work (0.140), failure (0.138), redirect (0.127), resource (0.127), trigger (0.120), return (0.119), validatoradapter (0.105), issue (0.105), construct (0.109), call (0.099)

Table 6.11: Top-10 discriminative terms based on the correlation value for each key feature of the Firefox project

Projects	Features	Top-10 discriminative terms
Firefox	CB	handler (0.231), javascript (0.201), bug (0.197), filter (0.188), cover (0.188), read (0.188), protocol (0.142), bind (0.142), event(0.142), html (0.138)
	TC	javascript (0.291), intermittent (0.261), browser (0.251), expect (0.213), chrome (0.192), uncaught (0.181), test (0.177), add(0.162), type (0.128), call (0.127)
	STR	browser (0.255), javascript (0.246), test (0.230), intermittent (0.226), content (0.180), remove (0.157), user (0.146), window (0.146), unexpect (0.139), perform (0.127)
	ST	browser (0.598), javascript (0.590), intermittent (0.576), test (0.391), expect (0.369), uncaught (0.351), crash (0.35), error (0.256), unexpect (0.243), np (0.23)
	EB	javascript (0.346), intermittent (0.344), browser (0.326), test (0.251), content (0.238), uncaught (0.216), error (0.205), chrome (0.191), build (0.180), perform (0.170)

Table 6.12: Top-10 discriminative terms based on the correlation value for each key feature of the Thunderbird project

Projects	Features	Top-10 discriminative terms
Thunderbird	CE	bust (0.231), port (0.225), bug (0.204), source (0.188), define (0.188), http (0.188), script (0.188), error (0.161), folder (0.146), remove (0.145)
	TC	unexpect (0.350), fail (0.345), test (0.342), javascript (0.311), mozmil (0.283), xpcshel (0.243), toolkit (0.182), component (0.160), mail (0.157), content (0.157)
	STR	test (0.292), unexpect (0.251), javascript (0.234), xpcshel (0.174), mozmil (0.174), delete (0.173), filter (0.169), bug (0.167), fail (0.160), remove (0.149)
	ST	crash (0.448), test (0.393), mozmil (0.326), javascript (0.311), unexpect (0.304), fail (0.266), mail (0.211), dbview (0.175), init (0.175)
	EB	test (0.309), unexpect (0.266), javascript (0.256), fail (0.243), component (0.228), mozmil (0.223), crash (0.212), failure (0.168), build (0.166), error (0.145)

We also find that, in some cases, reporters include very short summaries. For example, the description of the bug report [1] contains Stack Trace, Test Case, and Expected Behavior. During the model evaluation (testing), our prediction model should correctly predict, Stack Trace, Test Case, and Expected Behavior for this bug report. The summary of this report is "MinaConsumerTest Failure," which contains only two terms, "MinaConsumerTest" and "failure." In the table 6.11 and 6.12, we find that the term "failure" appears among the top-10 most discriminative terms for Test Case. Thus, our model might predict Test Case because it is trained using the summaries of the bug reports. For Stack Trace and Expected Behavior,

[1] https://issues.apache.org/jira/browse/CAMEL-795

our models might provide an incorrect prediction because the summaries do not contain terms correlated with Stack Trace and Expected Behavior. In another example, the description of the bug report [2] contains Steps to Reproduce, Stack Trace, and Code Example. Thus, during the model evaluation, our prediction model should predict these features correctly. The summary of the bug report is "NPE from came:run with below route codes." After removing stop words, the summary contains the terms "NPE (NullpointerExpection)," "Came," "run," "route," and "code." In the table 6.11 and 6.12, we find "NPE," "run," and "route" appear in the top 10 most discriminative terms for Stack Trace, Steps to Reproduce, and Code Example, respectively. That might help to predict these features. Therefore, the performance of our models might rely on the terms contained in the summaries of the bug reports.

To demonstrate the usefulness of our prediction models in practice, we introduce two cases from Camel project such as bug reports-CAMEL-4171 [3] and CAMEL-2909 [4]. In the case of CAMEL-4171, the reporter, Sergey Zhemzhitsky, provided Observed Behavior, Stack Trace, and Code Example as the unstructured features in the description of the bug report. When this bug report was assigned to the developer, Claus Ibsen, to fix, he failed to reproduce the bug and asked for the additional feature. One day later, another developer, Freeman Fang, made a clarification question about the bug. Even after four months, the reporter did not respond with the requested feature. In the meantime, a new version was released. Then, the developer made a follow-up question i.e.," *Any update? Did you try with a later release or created a unit test".* Later, the bug was also identified in the new version. One year and 3 months later, the developer received the requested feature and finally resolved the bug. In the evaluation phase of our prediction model, we see that our model correctly predicted "Steps to Reproduce" as the required feature for this bug.

In the case of CAMEL-2909, the reporter, Max Matveev, submitted this report on March 02, 2012. When this bug report was assigned to the developer, Claus Ibsen to fix, he requested to the reporter to provide Stack Trace and Code Example as additional required features to fix the bug. In this case, our model

[2]https://issues.apache.org/jira/browse/CAMEL-550
[3]`https://issues.apache.org/jira/browse/CAMEL-4171`
[4]`https://issues.apache.org/jira/browse/CAMEL-2909`

correctly predicted Code Example successfully. However, our model made the wrong prediction for the Stack Trace.

Another purpose of our quantitative analysis is to determine the best-performing classification technique for building the prediction model. Our significance test (sec Table 6.7) shows there is no single best-performing technique. However, NBM performs comparatively better than the other techniques do. We do not know exactly why NBM outperforms the other techniques. One possible reason is our study design. We build prediction models based on the summaries of the bug reports. The summary contains a limited number of terms. SVM performs better in full-length documents, whereas NBM performs better in short documents [83]. NB assumes that each of the features is conditionally independent [84]. However, in reality, this assumption of independence is rarely true. Instead, NBM uses a multinomial distribution and works better than NB [83, 85]. In spite of its better prediction performance, NBM is a simple and fast technique for training and testing the model [84]. The purpose of building key features prediction model is to develop an automated features recommendation tool that works in real-time. Thus, a simple and quick technique is essential. Therefore, NBM may be effective in building a classification model for predicting key features.

Table 6.13: F1-Scores to predict the key features in the Cross-Project setting for the Apache projects

Projects	Code Example	Test Case	Steps to Reproduce	Stack Trace	Expected Behavior
Camel – > Derby	0.536	0.409	0.370	0.271	0.505
Camel – > Wicket	0.664	0.328	0.308	0.183	0.572
Derby – > Camel	0.485	0.444	0.395	0.271	0.495
Derby – > Wicket	0.496	0.475	0.374	0.301	0.590
Wicket – > Camel	0.647	0.311	0.402	0.137	0.609
Wicket – > Derby	0.573	0.375	0.382	0.274	0.536

Our quantitative analysis has shown that NBM outperforms over other classification techniques to predict the key features within-project setting. We want to further investigate whether our models with NBM can work for predicting the key features in the cross-project setting. To build prediction models in cross-project setting, we use one dataset for the training the model and another dataset for the testing the performance of the model. The table 6.13 shows the f1-scores of predicting the key features in the cross-project setting. We notice that in most of the cases, our models achieve in the cross-project setting only a bit worse than those achieved in the within-project setting. For example, the best f1-score of our models in the within-project for predicting Test Case of Camel project is 0.410 while in the cross-project setting the f1-score is 0.409 (e.g., Camel->Derby setting). Therefore, we conclude that our models can work for predicting the key features in the cross-project setting.

In our qualitative analysis, we found that the features missing from the initial submission are unstructured features. Reporters provide these unstructured features in the descriptions of the bug reports as unstructured natural language text. Existing studies have revealed 10 unstructured features that are important to developers when fixing bugs. However, these features are not all equally important for all types of bug reports [5, 14, 86]. By examining the bug-fixing activities in our qualitative analysis, we know which unstructured features are required to fix each bug report. Thus, machine-learning techniques could help to predict essential unstructured features when writing new bug reports by leveraging historical bug-fixing activities. Therefore, we motivate building a prediction model based on the summary text so that reporters of new bug reports might know which key features should be included. Thus, in order to help reporters, we build prediction models using the NB, NBM, KNN, and SVM text-classification techniques, based on the summary text. We evaluate our models using the bug reports of the Camel, Derby, Wicket, Firefox, and Thunderbird projects. Our models achieve promising f1-scores when predicting key features, except for Stack Trace of the Wicket project and Code Example of Firefox project. Given that our models are the first automated support of their kind, such performance can make a difference when writing bug reports.

6.5.2 Implications

For Reporters:

Our models achieve promising f1-scores and accuracy to predict key features based on the summary text of the bug reports. The reporters of the new bug reports can know whether they need to provide a certain key feature to make a good bug report at the initial submission. It helps the reporters especially novice and end-users to write a good bug report by providing the minimum number of key features. Our models need to be trained on historical bug reports. Practitioners can easily collect historical bug reports from issue tracking systems such as JIRA and Bugzilla. Therefore, our model is practical and can be of much benefit.

For Researchers:

Reporters might need more accurate and easier models, although our models achieve a promising f1-Score to identify key features. To predict the key feature, we apply four popular classification techniques to build prediction models. To mitigate the class imbalance learning problems, we apply the basic imbalance learning strategy since our datasets have a class imbalance learning problems. However, more sophisticated techniques might provide better results. This highlight an opportunity for further research to extend, customize or invent advanced techniques to develop more accurate models.

6.6 Threats to Validity

In this section, we discuss some potential threats to the validity of our study, based on the guidelines proposed by [87].

6.6.1 External validity

Threats to external validity relate to the generalizability of our results. We build and test our models using five OSS projects from two ecosystems. Thus, our results may not generalizable to all software systems especially proprietary systems.

6.6.2 Internal validity

Our concerns related to internal threats are the experimental bias and errors. We constructed training and testing data sets with which to build and test our prediction models. We found that our data sets have a class imbalance problem. To mitigate this problem, we applied SMOTE, a popular class imbalance learning technique. In addition, to reduce the training data set selection bias, we applied 10-fold cross-validation and repeated the experiment 10 times to report the average performance. Our prediction models are trained and tested based using the summaries of the bug reports. Therefore, they rely on well-written summaries. An inaccurate summary may degrade the performance of our approach.

6.7 Chapter Summary

Our goal is to propose an approach that predicts whether reporters should provide certain features in their reports. To achieve this goal, we develop an approach to support reporters while writing new bug reports. We build classification models to predict the key features using four popular text-classification techniques. Our models are trained using the summaries of the bug reports so that reporters may know which features to provide in the descriptions of new bug reports. We evaluate our prediction models using the bug reports of the Camel, Derby, Wicket, Firefox, and Thunderbird projects. Our models achieve the best f1-score for Code Example, Test Case, Steps to Reproduce, Stack Trace and Expected Behavior of 0.70 (Wicket), 0.70 (Derby), 0.70 (Firefox), 0.65 (Firefox), and 0.76 (Firefox), respectively, which are promising. Our comparative study of the different classifications techniques reveals that NBM outperforms the other techniques when predicting key features. We also compare the performance of our models with the baseline models. The results show that our models provide a 12–115% improvement over the baseline models. Our experimental results also show that our best performing model (NBM) can work for predicting the key features in the cross-project setting.

7 Key Features Recommendation Model

The goal of this chapter is to develop a key features recommendation model to suggest features that reporters should be provided in the description of the bug reports. To achieve the goal, we develop a recommendation model by leveraging historical bug fixing knowledge and machine learning techniques. Our model, first, identify topN (N=1,3,5,10) similar bug reports of a new bug report based on the summary text of the bug reports. Then, it scans the topN similar bug reports and extract key features. Finally, our model recommends key features to the reporters based on the topN similar bug reports.

7.1 Introduction

Bug reports play an important role in all phases of the bug fixing process. In OSS projects, most bug reporters work voluntarily. In many cases, reporters submit incomplete or inaccurate bug reports that affect the different phases of the bug fixing process. Thus, tools support is essential to improve the bug fixing process. Existing researchers focus on building tool to support reporters by notifying unstructured features that are missing in the description of the bug reports [1, 21]. However, not all unstructured features are necessarily appropriate for all bug reports [5]. Thus, existing tools may provide unnecessary notifications to the reporters that might increase the complexity of bug reporting. In order to support bug reporting, in chapter 6, we build classification models to predict whether the certain key features reporters should provide in the description of the bug reports. We build a separate classification model for each key features based on the summary text of the bug reports. Our models achieve promising F1-scores to predict key features. However, in a real-time web environment, running multiple models might be time-consuming. Thus, a simpler model is essential to support reporters in real time.

To address the above issue and to develop a simpler model, in this chapter, we proposed a key features recommendation model to suggest features that reporters should provide in the description of the bug reports. In recent years, information retrieval (IR) and machine learning based automatic techniques has gained popularity to identify similar bug reports [28, 49, 50] and duplicate bug reports [34, 51, 52]. Rocha et al. [49] used the summary text of the bug reports to train their model and perform well to identify pending bug reports. We also utilize similar bug concept because bug-tracking systems for large open-source software (OSS) projects contain a large number of bug reports of fixed bugs. For each bug, it contains initial bug reports, developers and reporters activities during fixing, patches that fixed bugs, etc. Thus, examining historical bug reports can be a good way to understand which features developers require to fix bugs. Our intuition is developers may require similar features to fix similar bugs. Thus, in our approach, first, we calculate the similarity scores of the bug reports of the fixed bugs for the new bug reports based on the summary text. Then, we select the topN (N=1,2,3,......N, means top N similar documents) similar bug reports based on the similarity score. Finally, our model recommends the key features based on the key features of the topN similar bug reports. To evaluate the effectiveness of our recommendation model, we conduct an experiment on the bug reports of three projects (Camel, Derby, and Wicket) from the Apache ecosystem and two projects (Firefox and Thunderbird) from the Mozilla ecosystem. The experimental results show that our key features recommendation model can successfully recommend key features that reporters should provide in the description to make good bug reports.

7.1.1 Chapter Organization

The remainder of the chapter is organized as follows. Section 7.2 describes background and definition of related terms and techniques. Section 7.3 describes the design of our proposed key features recommendation model. Section 7.4 presents and discusses the experimental results. Section 7.5 discusses the limitation of our model. Finally, section 7.6 summarizes this work.

7.2 Background and Definition

All IR (information retrieval) based techniques need to extract and and preprocess textual contents from the historical bug reports. First, we extract the summary text and features from the description of the bug reports. Next, after the textual contents from the bug reports are obtained, we need to preprocess them. The purpose of text preprocessing is to standardize words in bug reports. There are three main steps: text normalization, stopword removal, and stemming. The following is the description of each step.

- **Text normalization:** In this step, punctuation marks and special symbols are deleted from documents (i.e., bug reports). Then, documents are split into constituent words. The identifiers are split into smaller words following the Camel casing convention (e.g., "ApacheCamel" is split into "Apache" and "Camel").

- **Stopword Removal:** In this step, English stopwords are deleted from documents (i.e., bug reports). These words frequently appear in many documents. Thus, they are not too helpful to differentiate a document from the other ones.

- **Stemming:** In this step, we transform English words to their root form. For example, "connection", "connecting", and "connected" are all reduced by stemming to connect.

.

There are many IR based techniques that have been employed for triaging and localizing bugs. We highlight a popular IR based technique namely Vector Space Model (VSM).

Vector Space Model(VSM): VSM is a widely used technique in traditional information retrieval. Several approaches [59, 63, 66] have adapt VSM to bug traiging and localizing software bugs. In VSM, each bug report is represented as an n-dimensional vector, where n is the number of unique index terms appearing in all the documents (d) and a new bug report is represent as a query (q), and w_t is the weight of the $i-th$ index term in the vector $< w_1, w_2, ..., w_n >$ defined as follows:

$$w_t \in d = tf_{td} \times idf_t \tag{7.1}$$

In equation 7.1, tf refers to the frequency of index term occurrences in a document and *idf* refers to the frequency of index term occurrences over the entire collection of documents. Among many variations of weights, the logarithmic variant was used because it can lead to better performance. A typical formula for tf and idf are shown in equation 7.2.

$$tf(t,d) = log(f_{td}) + 1 \tag{7.2}$$

$$idf(t) = log(\frac{N}{1+n_t}) \tag{7.3}$$

where t represents an index term, d represents a particular document, f_{td} is the number of term t occurs in document d, N is the total number of documents, and n_t is the number of documents in which term t occurs. After transforming bug reports into vectors, we calculate the degree of similarity between a given bug report and historical bug reports as shown in equation 7.4.

$$Similarity(q,d) = \frac{\vec{V_q} \cdot \vec{V_d}}{|\vec{V_q}||\vec{V_d}|} \tag{7.4}$$

7.3 Design of our Proposed key features recommendation model

7.3.1 Dataset Preparation

In order to validate our recommendation model, we use the bug reports of the three projects (Camel, Derby, and Wicket) from the Apache ecosystem and two projects (Firefox and Thunderbird) from the Mozilla ecosystem. We discuss in

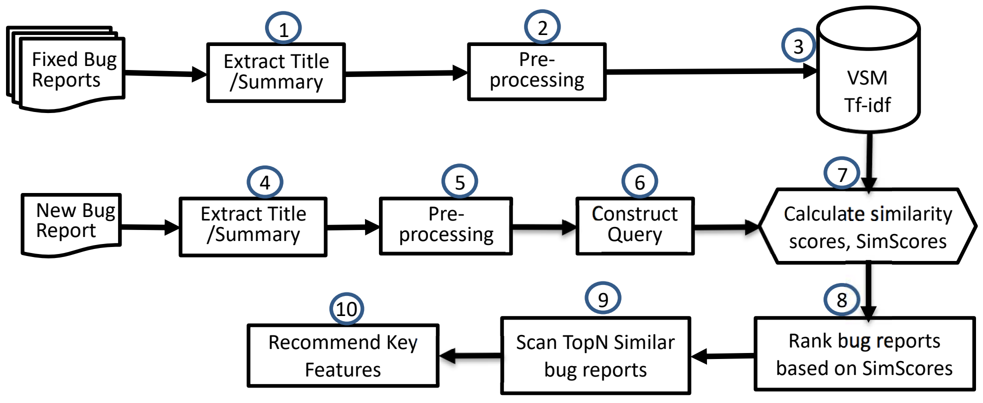


Figure 7.1: The overall structure of our recommendation model

details about dataset preparation especially, projects and bug reports section in chapter 4 (see section 4.2.1). We use the summary of the bug reports for building vector space model (VSM) and calculate similarity scores. For recommending key features, we use extracted features from the description and comments of the bug reports. Therefore, we prepare datasets for each project based on the summary and extracted features from the description and comments of the bug reports to evaluate the performance of our recommendation model.

7.3.2 Approach

The figure 7.1 depicts the overall structure of our key features recommendation model. In order to develop a key features recommendation model, we extract the title/summary from the fixed bug reports (i.e., marked as FIXED) and build training corpus. After tokenizing the training corpus, we apply pre-processing techniques, remove stop words, and convert terms into their root forms. Then, we build a TF-IDF vector space model for the training corpus. For the new bug report, we extract the title/summary from the initial bug report and apply similar pre-processing techniques similar way as the training corpus. We find the most similar fixed bug reports of the new bug report based on the cosine similarity scores. After that, it scans the description and comments of the most similar bug reports and extract the key features. Finally, our model recommends key features to reporters so that they can make a good quality description of a new bug report by providing the minimum number of key features.

7.3.3 Model Evaluation

To evaluate the performance of our prediction models, we use a traditional evaluation metric, namely accuracy. This metric is commonly used to evaluate classification performance [72] and can be derived from a confusion matrix. A confusion matrix lists all four possible classification results. If a feature of a bug report is present and predicts correctly, it is a true positive (Tp). If it predicts incorrectly, then it is a false positive (Fp). Similarly, there is a false negative (Fn) and a false positive (Fp) outcome. Based on Tp, Fp, Fn, and Tn, we calculate the accuracy.

Accuracy is the proportion of correctly recommended features in terms of total recommendation:
$Accuracy = \frac{Tp+Tn}{Tp+Fp+Fn+Tn}$.

In binary classification/prediction precision or recall is used to measure how accurately classify or predict a particular class (positive or negative). In our case (features recommendation), the accurate recommendation is important. Thus, we use accuracy to see how accurately our model can recommend features.

7.4 Evaluation Result and Discussion

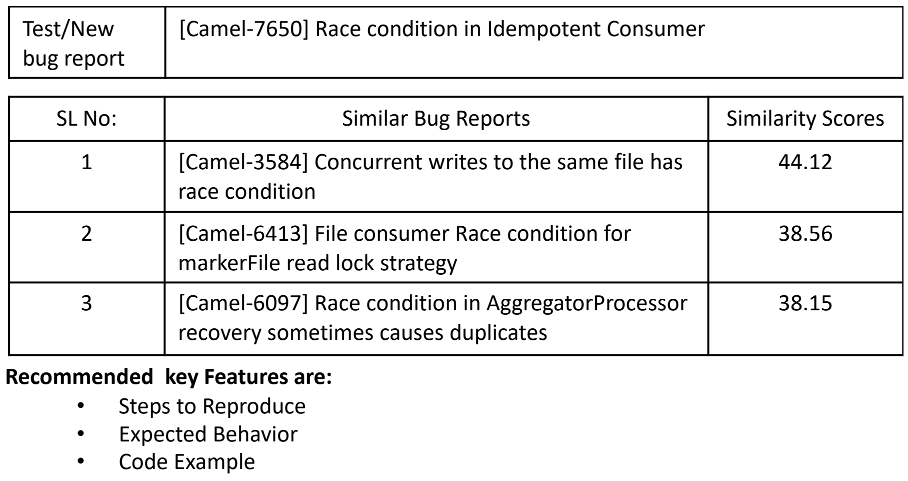
Approach: In our manual analysis (see Chapter 4), we identify reported unstructured features from the description and comments for each bug reports. Thus, we know the key features that were required to fix each bug. This required features set, we consider as ground truth for the evaluation of our recommendation model. In the testing phase, we compare the recommendation output with ground truth for each key feature to calculate the recommendation accuracy. For instance, B_1 is one of the bug reports of test dataset and “Steps to Reproduce” is the required feature for fix B_1. If our model recommended “Steps to Reproduce” correctly then the recommendation accuracy is 100% for the B_1. Otherwise, the recommendation accuracy is 0%. Similarly, we calculate recommendation accuracy for each feature of each testing bug report and present the average recommendation accuracy in

Table 7.1: Recommendation accuracy for Top 3 of our model for different projects

Features	Projects				
	Camel	Derby	Wicket	Firefox	Thunderbird
Code Example	0.57	0.78	0.63	0.89	0.81
Test Case	0.79	0.62	0.59	0.60	0.71
Steps to Reproduce	0.64	0.67	0.72	0.62	0.62
Stack Trace	0.86	0.75	0.90	0.91	0.85
Expected Behavior	0.59	0.63	0.61	0.65	0.69

the table 7.1.

Results: We see from the table 7.1 that the recommendation accuracy for Top 3 of our model is promising. For example, our recommendation model achieves the highest accuracy for Code Example is 0.89 for the Firefox project. It means that our recommendation model can accurately recommend Code Example for the 89% test cases of the Firefox project. In the case of Stack Trace, our model can accurately recommend 91% test cases of the Firefox project. These indicate that our model can effectively recommend key features that are useful for the developers to fix the bug. Existing tools focus on notifying features that are missing in the description of the bug reports. However, not all unstructured features are equally important for each bug report to fix. Thus, existing tools may provide unnecessary notifications. Our model recommends only those features that were useful to fix similar types of bugs. The fiqure 7.2 shows a use case of our recommendation model. The Camel-7650 is a bug report of the Camel project. Camel-3584, Camel-6413, and Camel-6097 are the most similar bug report of the Camel-7650. We see that Steps to Reproduce, Expected Behavior, and Code Example were the useful features to fix these three bugs. Thus, our model recommends these three features for the Camel-7650. Based on the recommended features, reporters can make a good bug report by providing the minimum number of unstructured features.



Test/New bug report	[Camel-7650] Race condition in Idempotent Consumer

SL No:	Similar Bug Reports	Similarity Scores
1	[Camel-3584] Concurrent writes to the same file has race condition	44.12
2	[Camel-6413] File consumer Race condition for markerFile read lock strategy	38.56
3	[Camel-6097] Race condition in AggregatorProcessor recovery sometimes causes duplicates	38.15

Recommended key Features are:

- Steps to Reproduce
- Expected Behavior
- Code Example

Figure 7.2: An example recommendation of our model

7.5 Usages of our model

7.5.1 Integration of our model with existing bug tracking system

Our propose features recommendation model can integrate with the Jira/Bugzilla issue tracking system. The figure 7.3 shows the overall architecture of our proposed integration. In the web interface, a reporter writes the summary of a bug report. An Ajax event fire after writing the summary of a bug report and start preprocessing in the server side. Then, identify topN similar bug reports based on the summary and generate recommended features set by scanning the top similar bug reports. Finally, our model sends the recommendation to the reporters on the web interface.

7.5.2 Applicability of our model in high impact bug

Our analysis results the bug reports of high impact bugs show that the degree of key features varies among the different types of high impact bugs. High impact bugs should be fixed as early as possible because of its impact on the

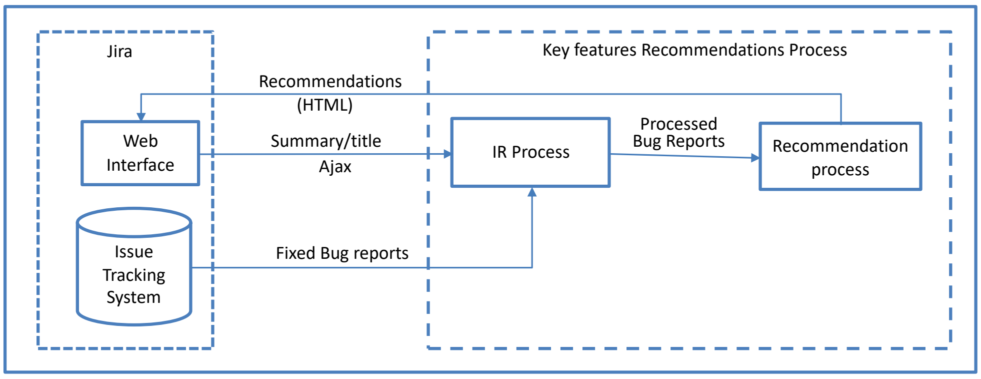


Figure 7.3: The overall architecture of our recommendation model

software development process, product, and end-users. Thus, the accurate key features recommendation is very essential. Experimental results show that our recommendation model can accurately recommend key features. To train and test our prediction and recommendation models, we did not consider types of high impact bugs in this research. In chapter 6, we see that our models can accurately predict key features in both within projects and cross projects setting. These suggest that our model can also apply to high impact bugs. If you train and test our model within the same type of high impact bug then reporters may get more accurate features recommendation.

7.5.3 Implications

For Reporters

Evaluation results show that our proposed features recommendation model achieve promising accuracy to recommend key features. Bug reporters can get accurate features recommendation that is important for the developers to fix bugs based historical bug fixing insights. Thus, reporters can make good bug reports by providing the minimum number of features.

Bug Tracking System

Our empirical study reveals a large percentage of bug reports do not contain sufficient features at the initial submission. This indicates that bug reporters need real-time support to make good bug reports. Our model needs to be trained on historical bug reports. Practitioners can easily collect historical bug reports from issue tracking systems such as JIRA and Bugzilla. Therefore, our model is practical and can be of much benefit. By incorporating our model with the existing bug reporters system, bug reporters would get real-time support to make good reports.

7.6 Limitations

In this section, we discuss some potential threats to the validity of our study, based on the guidelines proposed by [87]. Threats to external validity relate to the generalizability of our recommendation model. We test our model using the sample bug reports of five OSS projects from two ecosystems. The sample bug reports may not represent the actual picture of the project. Thus, our results may not generalizable to all software systems especially for the proprietary systems. To identify similar bug reports of the new bug report, we use the summary text. In some cases, reporters make a very short summary. Thus, some similar bug reports may get a low similarity score because of the short summary text since the similarity score is calculated based on the textual similarity. Therefore, in the worst case, our recommendation model may produce an inappropriate recommendation.

7.7 Chapter Summary

This chapter proposed a novel approach called a key features recommendation model to suggest features that reporters should provide in the description of the bug reports. Our model utilizes the description and comments of the bug reports to generate a key features recommendation. When a reporter writes the summary of a new bug report, our recommendation model first identifies the most similar bug reports based on the summary text of the bug report. Then, our model scans the description and comments of the topN similar bug reports and recommend key features based on the topN similar bug reports. We evaluate the accuracy of our recommendation model using the bug reports of three projects (Camel, Derby, and Wicket) from the Apache ecosystem and two projects (Firefox and

Thunderbird) from the Mozilla ecosystem. The experimental results show that our model can accurately recommend key features. The following are the some of the challenges that need to address to improve the performance of our proposed a key features recommendation model.

- **Accurately Identify Similar Bug Reports:** We use the title/summary text to calculate the similarity score between a new bug report with historical bug reports. In some cases, bug reports of fixed bugs contain a very short summary. It affects the similarity scores. Thus, we plan to apply different machine learning techniques such as Word Embedding and Word2Vec techniques.

- **Validation of Our Recommendation Model:** Initially, we use accuracy to evaluate the performance of our recommendation model. We have a plan to evaluate our recommendation model using other evaluation metrics.

- **Automatically Extracting Features from Bug Reports:** We are planning to suggest the actual features so that reporters can make a good bug report by slightly modifying the recommended features. However, we are facing challenges to automatically extracting features from the bug reports of fixed bugs.

8 Conclusion

Bug reports are the primary means through which developers triage and fix bugs. It plays an important role in all phases of the bug-fixing process. Prior research shows that there is a clear mismatch between the features that developers would wish to appear in a bug report, and the features that actually appear. They also found that bug reports are neither complete nor accurate, and often do not provide all the features that developers find useful when fixing bugs. In an effort to improve bug management process, the goal of this research is to understand key features that make a good bug report and to develop an automatic key features recommendation model to support reporters for making a good bug report. To achieve our goal, we first perform a qualitative analysis of five OSS projects to investigate the key features of a bug report by examining bug-fixing activities. We also conduct an empirical study on the bug reports of six types of high impact bugs of the Apache Camel project. Our qualitative analysis reveals that Steps to Reproduce, Test Case, Code Example, Stack Trace, and Expected Behavior are the additional required features that developers often requested during bug fixing. Through qualitative analysis, we identify five key features (i.e., Steps to Reproduce, Test Case, Code Example, Stack Trace, and Expected Behavior) that reporters often miss in their initial bug reports and developers require them for fixing bugs. Our empirical study on the bug reports of high impact bugs reveals that the degree of key features varies among the different types of high impact bugs, however, four key features are almost the same. Our statistical analyses reveal that the additional requirement for the key features during bug fixing significantly affect the bug-fixing process.

In an effort to support the reporters to make a good bug report, we first develop classification models to predict the key features using four popular text-classification techniques. Our models are trained using the summaries of the bug reports so that reporters may know which features to provide in the descriptions of new bug reports. We evaluate our prediction models using the bug reports of the Camel, Derby, Wicket, Firefox, and Thunderbird projects. Our models achieve the best f1-score for Code Example, Test Case, Steps to Reproduce, Stack Trace and Expected Behavior of 0.70 (Wicket), 0.70 (Derby), 0.70 (Firefox), 0.65 (Firefox), and 0.76 (Firefox), respectively, which are promising. Our comparative study of the different classifications techniques reveals that NBM outperforms the other techniques when predicting key features. In order to investigate whether our best performing model is applicable in the cross-projects setting to predict key features, we train our model by one projects' dataset and evaluate the performance of the model by another projects' dataset. The result shows that our model can work successfully in the cross-projects setting to predict key features. We also compare the performance of our models with the baseline models. The results show that our models provide a 12–115% improvement over the baseline models. Then, we proposed a novel approach called key features recommendation model to suggest features that reporters should provide in the description of the bug reports. The experimental results show that our model can accurately recommend key features.

References

[1] Thomas Zimmermann, Nachiappan Nagappan, Philip J. Guo, and Brendan Murphy. Characterizing and predicting which bugs get reopened. In *34th International Conference on Software Engineering*, ICSE ’12, 2012.

[2] Sascha Just, Rahul Premraj, and Thomas Zimmermann. Towards the next generation of bug tracking systems. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, VL/HCC ’09, pages 82–85, 2008.

[3] Nicolas Bettenburg, Sascha Just, Adrian Schroter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT ’08/FSE-16, 2008.

[4] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. Information needs in bug reports: Improving cooperation between developers and users. In *ACM Conference on Computer Supported Cooperative Work*, CSCW ’10, 2010.

[5] Steven Davies and Marc Roper. What’s in a bug report? In *8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM ’14, 2014.

[6] Jorge Aranda and Gina Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. In *31st International Conference on Software Engineering*, ICSE ’09, 2009.

[7] Tommaso Dal Sasso, Andrea Mocci, and Michele Lanza. What makes a satisficing bug report? In *IEEE International Conference on Software Quality, Reliability and Security*, QRS ’16, 2016.

[8] Tao Zhang, Jiachi Chen, He Jiang, Xiapu Luo, and Xin Xia. Bug report enrichment with application of automated fixer recommendation. In *25th International Conference on Program Comprehension*, ICPC '17, 2017.

[9] Apache Camel project. Issue tracking system, jira. url: https://issues.apache.org/jira/browse/camel-4820.

[10] Apache Camel project. Issue tracking system, jira. url: https://issues.apache.org/jira/browse/derby-893.

[11] Apache Camel project. Issue tracking system, jira. url: https://issues.apache.org/jira/browse/wicket-1159.

[12] Apache Camel project. Issue tracking system, jira. url: https://issues.apache.org/jira/browse/ambari-9083.

[13] Philip J. Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. Characterizing and predicting which bugs get fixed: An empirical study of microsoft windows. In *32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, 2010.

[14] Md. Rejaul Karim, Akinori Ihara, Xin Yang, Hajimu Iida, and Kenichi Matsumoto. Understanding key features of high-impact bug reports. In *8th International Workshop on Empirical Software Engineering in Practice*, IWESEP '17, pages 53–58, 2017.

[15] Mona Erfani Joorabchi, Mehdi Mirzaaghaei, and Ali Mesbah. Works for me! characterizing non-reproducible bug reports. In *11th Working Conference on Mining Software Repositories*, MSR '14, 2014.

[16] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *28th International Conference on Software Engineering*, ICSE '06, 2006.

[17] Gerardo Canfora and Luigi Cerulo. Fine grained indexing of software repositories to support impact analysis. In *International Workshop on Mining Software Repositories*, MSR '06, 2006.

[18] Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports. In *34th International Conference on Software Engineering*, ICSE '12, 2012.

[19] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How long will it take to fix this bug? In *4th International Workshop on Mining Software Repositories*, MSR '09, 2009.

[20] Dongsun Kim, Yida Tao, Sunghun Kim, and Andreas Zeller. Where should we fix this bug? a two-phase recommendation model. *IEEE Transactions on Software Engineering*, 39(11), nov 2013.

[21] Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. Detecting missing information in bug descriptions. In *11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE '17, 2017.

[22] An open letter to github from the maintainers of open source projects. available online: https://github.com/dear-github/dear-github, 2016.

[23] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schroter, and Cathrin Weiss. What makes a good bug report? *IEEE Transactions on Software Engineering*, 36(5), 2010.

[24] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. Extracting structural information from bug reports. In *International Working Conference on Mining Software Repositories*, MSR '08, 2008.

[25] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. Security versus performance bugs: A case study on firefox. In *8th Working Conference on Mining Software Repositories*, MSR '11, 2011.

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

[27] Henrique Rocha, Marco Valente, Humberto Marques-Neto, and Gail C. Murphy. An empirical study on recommendations of similar bugs. In *23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, SANER '16, 2016.

[28] Xin-Li Yang, David Lo, Xin Xia, Qiao Huang, and Jian-Ling Sun. High-impact bug report identification with imbalanced learning strategies. *Computer Science and Technology*, 32:181–198, 2017.

[29] Xin Xia, David Lo, Weiwei Qiu, Xingen Wang, and Bo Zhou. Automated configuration bug report prediction using text mining. In *38th IEEE Annual International Computers, Software and Applications Conference*, COMPSAC '14, 2014.

[30] Yu Zhao, Feng Zhang, Emad Shihab, Ying Zou, and Ahmed E. Hassan. How are discussions associated with bug reworking?: An empirical study on open source projects. In *10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '16, 2016.

[31] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology Acquisition and Assistance Division*, 2002.

[32] Bug Tracking System. , jira, url: http://www.bugzilla.org/.

[33] Issue Tracking System. ,jira, url: https://issues.apache.org/.

[34] Chengnian Sun, David Lo, Xiaoyin Wang, Jing Jiang, and Cheng Khoo. A discriminative model approach for accurate duplicate bug report retrieval. In *32nd ACM/IEEE International Conference on Software Engineering*, ICSE '10, 2010.

[35] Apache Camel project. Issue tracking system, jira. url: https://issues.apache.org/jira/browse/camel-5860.

[36] Apache Camel project. Issue tracking system, jira. url: https://issues.apache.org/jira/browse/camel-5782.

[37] Likert Rensis. A technique for the measurement of attitudes, 1932.

[38] Andrew J. Ko, Brad A. Myers, and Duen Horng Chau. A linguistic analysis of how people describe software problems. In *Visual Languages and Human-Centric Computing*, VLHCC '06, 2006.

[39] Pieter Hooimeijer and Westley Weimer. Modeling bug report quality. In *22nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, 2007.

[40] Yutaro Kashiwa, Hayato Yoshiyuki, Yusuke Kukita, and Masao Ohira. A pilot study of diversity in high impact bugs. In *30th International Conference on Software Maintenance and Evolution*, ICSME '14, 2014.

[41] Emad Shihab, Audris Mockus, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. High-impact defects: A study of breakage and surprise defects. In *19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, 2011.

[42] Tse-Hsun Chen, Meiyappan Nagappan, Emad Shihab, and Ahmed E. Hassan. An empirical study of dormant bugs. In *11th Working Conference on Mining Software Repositories*, MSR '14, 2014.

[43] Emad Shihab, Akinori Ihara, Yasutaka Kamei, and Walid Ibrahim. Predicting re-opened bugs: A case study on the eclipse project. In *17th Working Conference on Reverse Engineering*, WCRE '10, 2010.

[44] 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 *12th Working Conference on Mining Software Repositories*, MSR '15, 2015.

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

[46] Harold Valdivia Garcia and Emad Shihab. Characterizing and predicting blocking bugs in open source projects. In *11th Working Conference on Mining Software Repositories*, MSR '14, 2014.

[47] Xia Xin, Lo David, Shihab Emad, Wang Xinyu, and Yang Xiaohu. Elblocker: Predicting blocking bugs with ensemble imbalance learning. *Information and Software Technology*, 61, 2015.

[48] Joel Spolsky. Joel on software blog.

[49] Henrique Rocha, Guilherme de Oliveira, Humberto Marques-Neto, and Marco Tulio Valente. Nextbug: a bugzilla extension for recommending similar bugs. *Journal of Software Engineering Research and Development*, 3(1), 2015.

[50] Shuo Wang and Xin Yao. Using class imbalance learning for software defect prediction. *IEEE Transactions on Reliability*, 62:434–443, 2013.

[51] Yuan Tian, Chengnian Sun, and David Lo. Improved duplicate bug report identification. In *16th European Conference on Software Maintenance and Reengineering*, CSMR '12, 2012.

[52] Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N Nguyen, David Lo, and Chengnian Sun. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *27th IEEE/ACM International Conference on Automated Software Engineering*, ASE '12, 2012.

[53] Xinli Yang, David Lo, Xin Xia, Lingfeng Bao, and Jianling Sun. Combining word embedding with information retrieval to recommend similar bug reports. In *2016 IEEE 27th International Symposium on Software Reliability Engineering*, ISSRE, '16, pages 127–137, 2016.

[54] Henrique Rocha, Guilherme de Oliveira, Humberto Marques-Neto, and Marco Tulio Valente. Nextbug: a bugzilla extension for recommending similar bugs. *Journal of Software Engineering Research and Development*, 2015.

[55] D. Hu, M. Chen, T. Wang, J. Chang, G. Yin, Y. Yu, and Y. Zhang. Recommending similar bug reports: A novel approach using document embedding model. In *25th Asia-Pacific Software Engineering Conference (APSEC)*, 2018.

[56] Ripon K. Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E. Perry. Improving bug localization using structured information retrieval. In *28th IEEE/ACM International Conference on Automated Software Engineering*, ASE'13, 2013.

[57] Wenjin Wu, Wen Zhang, Ye Yang, and Qing Wang. Drex: Developer recommendation with k-nearest-neighbor search and expertise ranking. In *18th Asia-Pacific Software Engineering Conference*, APSEC '11, 2011.

[58] Xin Ye, Hui Shen, Xiao Ma, Razvan Bunescu, and Chang Liu. From word embeddings to document similarities for improved information retrieval in software engineering. In *38th International Conference on Software Engineering*, ICSE '16, 2016.

[59] Xin Xia, David Lo, Xinyu Wang, and Bo Zhou. Accurate developer recommendation for bug resolution. *20th Working Conference on Reverse Engineering*, pages 72–81, 2013.

[60] Ferdian Thung, Tien-Duy B. Le, Pavneet Singh Kochhar, and David Lo. Buglocalizer: Integrated tool support for bug localization. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '14, 2014.

[61] Tien-Duy Le, Richard Oentaryo, and David Lo. Information retrieval and spectrum based bug localization: Better together. 09 2015.

[62] Amar Budhiraja, Kartik Dutta, Manish Shrivastava, and Raghu Reddy. Towards word embeddings for improved duplicate bug report retrieval in software repositories. In *2018 ACM SIGIR International Conference on Theory of Information Retrieval*, ICTIR '18, 2018.

[63] Xihao Xie, Wen Zhang, Ye Yang, and Qing Wang. Dretom: Developer recommendation based on topic models for bug resolution. In *8th International Conference on Predictive Models in Software Engineering*, PROMISE '12, 2012.

[64] Tien-Duy B. Le, Mario Linares-Vsquez, David Lo, and Denys Poshyvanyk. Rclinker: Automated linking of issue reports and commits leveraging rich contextual information. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, ICPC '15. IEEE Press, 2015.

[65] Xue Han, Tingting Yu, and David Lo. Perflearner: Learning from bug reports to understand and generate performance test frames. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18. ACM, 2018.

[66] W. Wen, T. Yu, and J. H. Hayes. Colua: Automatically predicting configuration bug reports and extracting configuration options. In *27th IEEE International Symposium on Software Reliability Engineering*, ISSRE '16, pages 107–116, 2016.

[67] M. M. Rahman, C. K. Roy, and D. Lo. Rack: Automatic api recommendation using crowdsourced knowledge. In *23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, SANER '16, pages 349–359, 2016.

[68] Masao Ohira, Ahmed E. Hassan, Naoya Osawa, and Kenichi Matsumoto. The impact of bug management patterns on bug fixing: A case study of eclipse projects. In *28th IEEE International Conference on Software Maintenance*, ICSM '12, 2012.

[69] Paige Rodeghero, Da Huo, Tao Ding, Collin McMillan, and Malcom Gethers. An empirical study on how expert knowledge affects bug reports. *Journal of Software Evolution and Process*, 28(7), July 2016.

[70] Michael Gegick, Pete Rotella, and Tao Xie. Identifying security bug reports via text mining: An industrial case study. In *7th Working Conference on Mining Software Repositories*, MSR '10, 2010.

[71] Adrian Nistor, Tian Jiang, and Lin Tan. Discovering, reporting, and fixing performance bugs. In *10th Working Conference on Mining Software Repositories*, MSR '13, 2013.

[72] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques.* O'Reilly Media, Inc., 2006.

[73] Aggarwal C. *Data Mining: The Textbook.* Springer Internation Publishing, 2015.

[74] Haibo He and Edwardo A. Garcia. Learning from imbalanced data. *IEEE Transactions on Knowledge and Data Engineering*, (9):1263–1284, 2009.

[75] Nitesh V. Chawla, Lawrence O. Bowyer, Kevin W.and Hall, and W. Philip Kegelmeyer. Smote: Synthetic minority over-sampling technique. *Artificial Intelligence Research*, 16, 2002.

[76] M. R. Karim and D. M. Farid. An adaptive ensemble classifier for mining complex noisy instances in data streams. In *International Conference on Informatics, Electronics Vision*, ICIEV, '14, 2014.

[77] Andrew McCallum and Kamal Nigam. A comparison of event models for naive bayes text classification. *in AAAI-98 Workshop on Learning for Text Categorization*, 752, 1998.

[78] Xindong Wu, Vipin Kumar, J. Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J. McLachlan, Angus Ng, Bing Liu, Philip S. Yu, Zhi-Hua Zhou, Michael Steinbach, David J. Hand, and Dan Steinberg. Top 10 algorithms in data mining. *Knowledge Information Systems*, 14(1), 2007.

[79] Davor Cubranic. Automatic bug triage using text categorization. In *16th International Conference on Software Engineering and Knowledge Engineering*, SEKE '04, 2004.

[80] Yasutaka Kamei, Akito Monden, Shinsuke Matsumoto, Takeshi Kakimoto, and Ken ichi Matsumoto. The effects of over and under sampling on fault-prone module detection. In *1st International Symposium on Empirical Software Engineering and Measurement*, ESEM '07, 2007.

[81] Minghui Zhou and Audris Mockus. Who will stay in the floss community? modeling participants' initial behavior. *IEEE Transactions on Software Engineering*, 41, 2015.

[82] Baljinder Ghotra, Shane Mcintosh, and Ahmed E. Hassan. A large-scale study of the impact of feature selection techniques on defect classification models. In *14th International Conference on Mining Software Repositories*, MSR '17, 2017.

[83] Sida Wang and Christopher D Manning. Baselines and bigrams: Simple, good sentiment and topic classification. In *50th Annual Meeting of the Association for Computational Linguistics: Short Papers - Volume 2*, ACL '12, 2012.

[84] Stuart Russell J. and Peter Norvig. Artificial intelligence: A modern approach (2nd). *Artificial Intelligence and Machine Learning Book*, 2003.

[85] Jason D. M. Rennie, Lawrence Shih, Jaime Teevan, and David R Karger. Tackling the poor assumptions of naive bayes text classifiers. In *20th International Conference on International Conference on Machine Learning*, ICML '03, 2003.

[86] Nor Shahida Mohamad Yusop, John Grundy, and Rajesh Vasa. Reporting usability defects: Do reporters report what software developers need? In *20th International Conference on Evaluation and Assessment in Software Engineering*, EASE '16, 2016.

[87] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2), apr 2009.

Appendix A

Understanding Key Features of a Bug Report

We use the following python code to download bug reports from the Jira issue tracking system.

```
import urllib.request
from urllib.error import URLError # the docs say this is the base
    error you need to catch
for i in range(0,10000):
    issue_id1='WICKET-'+str(i)
    url ="https://issues.apache.org/jira/si/jira.issueviews:issue-
    xml/"+issue_id1+'/'+issue_id1+'.xml'
    try:
        s=urllib.request.urlopen(url)
        contents = s.read()
    except URLError:
        print('an error occurred while fetching: "{}"'.format(url))
        continue # skip this url and proceed to the next
    file = open(issue_id1+'.xml', 'wb')
    file.write(contents)
```

We use the following python code to parse the bug reports and load into the MySQL database.

```
import os
from xml.etree import ElementTree
from xml.etree.ElementTree import ParseError
#from xml.dom import NotFoundErr
import mysql.connector
from mysql.connector import errorcode
try:
    cnn = mysql.connector.connect(
        user='root',
        password='Sdlab@jp14',
        host='localhost',
        database='jsme_journal'
    )
    print("It work fine")
except mysql.connector.Error as e:
    if e.errno == errorcode.ER_ACCESS_DENIED_ERROR:
        print("Something is wrong with username or password")
    elif e.errno == errorcode.ER_BAD_DB_ERROR:
        print("Database does not exist")
    else:
        print(e)
#file ='DERBY-5.xml'
count = 1
while(count < 11000):
    proName = "CAMEL"
    IssueId = proName+"-"+str(count)
    fileName = IssueId+".xml"
    file_status = os.path.isfile(fileName)
    count = count+1
    cursor = cnn.cursor()
    if file_status == True:
        dom = ElementTree.parse(fileName)
        channel = dom.findall('channel/item')
        for c in channel:
            strtitle = c.find('title')
            csummary = c.find('summary')
            summary = csummary.text
            cissue_type = c.find('type')
            cpriority=c.find('priority')
            cstatus=c.find('status')
            cresolution=c.find('resolution')
            creporter=c.find('reporter')
            cassinee=c.find('assignee')
            ccreate_date=c.find('created')
            cupdate_date=c.find('updated')
            cresolve_date=c.find('resolved')
            caffected_version=c.findall('version')
            cfixed_version=c.findall('fixVersion')
        print(cfixed_version)
        ccomponent=c.find('component')
        cvotes=c.find('votes')
        cwatches=c.find('watches')
        cdescription=c.find('description')
        comments=c.findall('comments')
        #cursor=cnn.cursor()
        strsql = ("INSERT INTO bug_reports(pName,issue_key,issue_type,status,priority,resolution,reporter,assinee,create_date,update_date,resolve_date,affected_version,fixed_version,component,votes,watches,summary,description,attachment) VALUES(%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s)")
        pName=proName
        issue_key=IssueId
        if cissue_type is None:
            issue_type="Null"
        else:
            issue_type=cissue_type.text
        if cstatus is None:
            status="Null"
        else:
            status=cstatus.text
        if cpriority is None:
            priority="Null"
        else:
            priority=cpriority.text
        if cresolution is None:
            resolution="Null"
        else:
            resolution=cresolution.text
        if creporter is None:
            reporter="Null"
        else:
            reporter=creporter.text
        if cassinee is None:
            assinee="Null"
        else:
            assinee=cassinee.text
        if ccreate_date is None:
            create_date="Null"
        else:
            create_date=ccreate_date.text
        if cupdate_date is None:
            update_date="Null"
        else:
            update_date=cupdate_date.text
        if cresolve_date is None:
            resolve_date="Null"
        else:
            resolve_date=cresolve_date.text
        if caffected_version is None:
            affected_version="Null"
        else:
            strev=""
            for ev1 in caffected_version:
               strev += ev1.text
               strev = strev+'/'
            affected_version=strev
        print(affected_version)
        if cfixed_version is None:
            print(cfixed_version)
            fixed_version="Null"
            print(fixed_version)
        else:
            strv=""
            print(strv)
            for v1 in cfixed_version:
                print(v1.text)
                strv +=v1.text
                strv=strv+'/'
            fixed_version=strv
            #fixed_version=cfixed_version.text
        print(fixed_version)

        if ccomponent is None:
            component="Null"
        else:
            component=ccomponent.text
        if cvotes is None:
                votes="0"
            else:
                votes=cvotes.text
            if cwatches is None:
                watches="0"
            else:
                watches=cwatches.text
            if csummary is None:
                summary="Null"
            else:
                summary=csummary.text
                summary=summary[0:1500]
            #print(summary)
            if cdescription is None:
                description="Null"
            else:
                description=cdescription.text
                if description is None:
                    description=description
                else:
                    description=description[0:6500]
                print(description)
                ##if len(description) >= 6500:
                #description=description[0:6500]
                #else:
                #     description=description
            attachment=0
            bug_report=(pName,issue_key,issue_type,status,priority,
resolution,reporter,assinee,create_date,update_date,resolve_date,
affected_version,fixed_version,component,votes,watches,summary,
description,attachment)
            cursor.execute(strsql,bug_report)
            cnn.commit()

            strsql = ("INSERT INTO bug_report_comments(issue_key,
comment_id,comments) VALUES(%s,%s,%s)")
            for c1 in comments:
                commentall=c1.findall('comment')
                for c2 in commentall:
                    issue_key=IssueId
                    if c2 is None:
                        continue
                    else:
                        cId=c2.get('id')
                        cmm=c2.text[0:5000]
                        comment=(issue_key,cId,cmm)
                        cursor.execute(strsql,comment)
                        cnn.commit()

            cursor.close()
            cnn.close
    else:
        print("File Not Found")
        continue

cnn.close
```

Appendix B

Investigating Key Features of High Impact Bugs (HIB) Report

We present here some of the comparative study results between high impact and normal bug reports. The figure 8.1 shows the distribution of bug fixing time among additional features request and non-request groups.

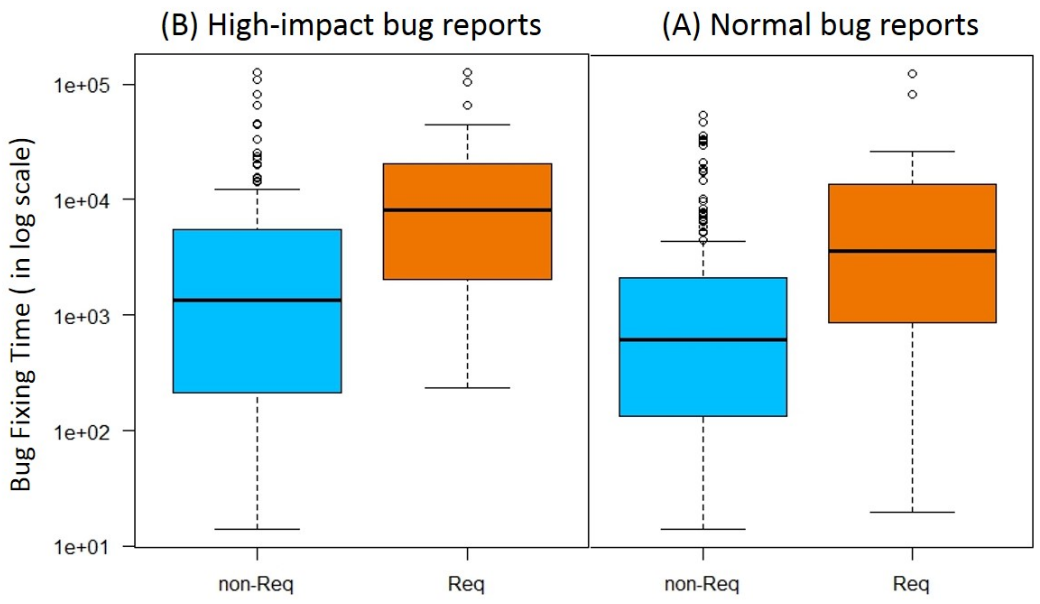


Figure 8.1: The Impact of additional request for features during bug fixing

The figure 8.2 shows the distribution of the bug reports among additional features requested and non-requested groups for high-impact bugs. The x-axis of the bar chart represents the high-impact bugs and the y-axis represents the percentages. We found that 0 to 29 percentages of total bug reports request for additional features during bug fixing. It indicates that reporters should be become more careful to file high-impact bug reports more accurately.

The figure 8.3 shows the distribution of the bug reports among additional features requested and non-requested groups for normal and others bug reports. The x-axis of the bar chart represents the types and the y-axis represents the percentages. We found that 75% of the total bug reports were requested for additional features in Incomplete bug reports. 10% and 24% of the total bug reports were requested for additional features in normal and unresolved bug reports respectively. It indicates that additional features request during bug fixing may be one of the reasons to become a bug reports incomplete and unresolved.

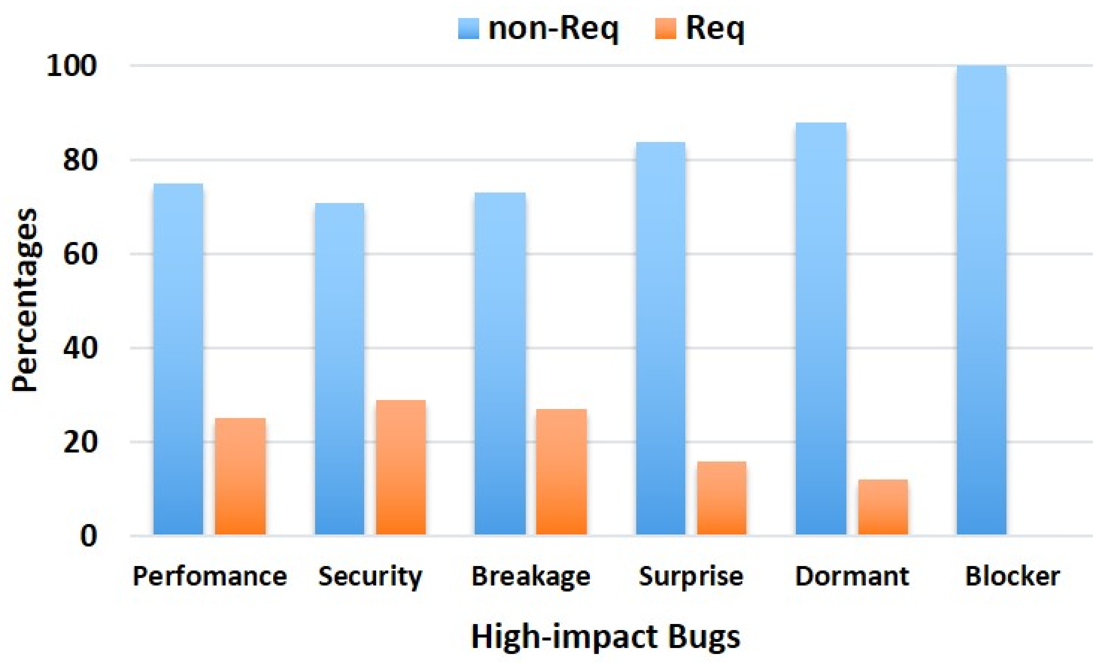


Figure 8.2: Distribution (additional Req vs non-Req) of bug reports in high-impact bugs

The figure 8.4 shows the distribution of overall bug fixing time among normal and high-impact bug reports.The x-axis of the bar chart represents the bug types and the y-axis represents the bug fixing time in log scale. In overall, bug fixing time for normal bug reports are slightly lower that the high-impact bug reports.

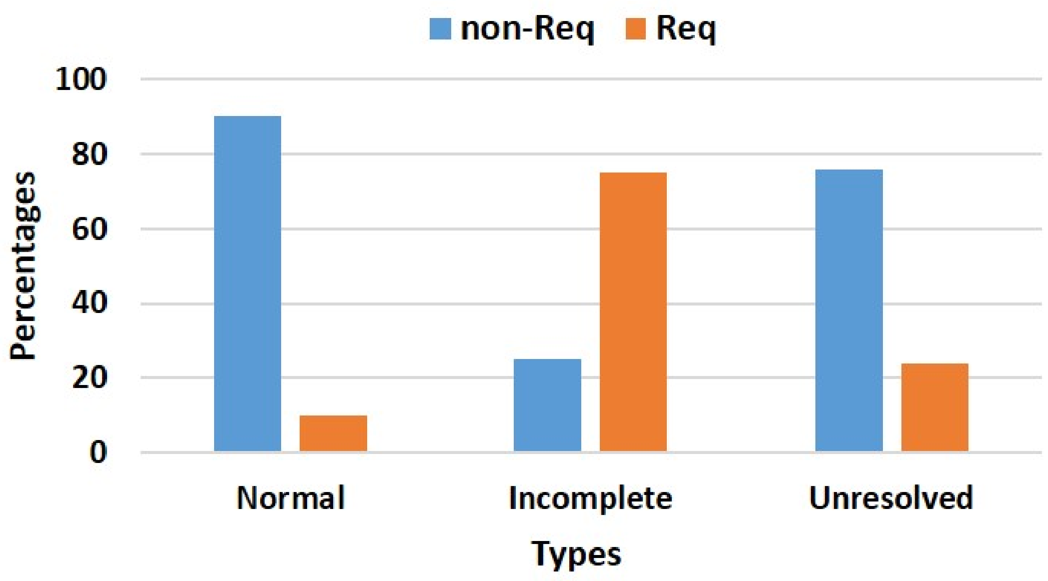


Figure 8.3: Distribution (additional Req vs non-Req) of bug reports in normal and others bugs

we have done significance test on bug fixing time among normal and high-impact bug reports.

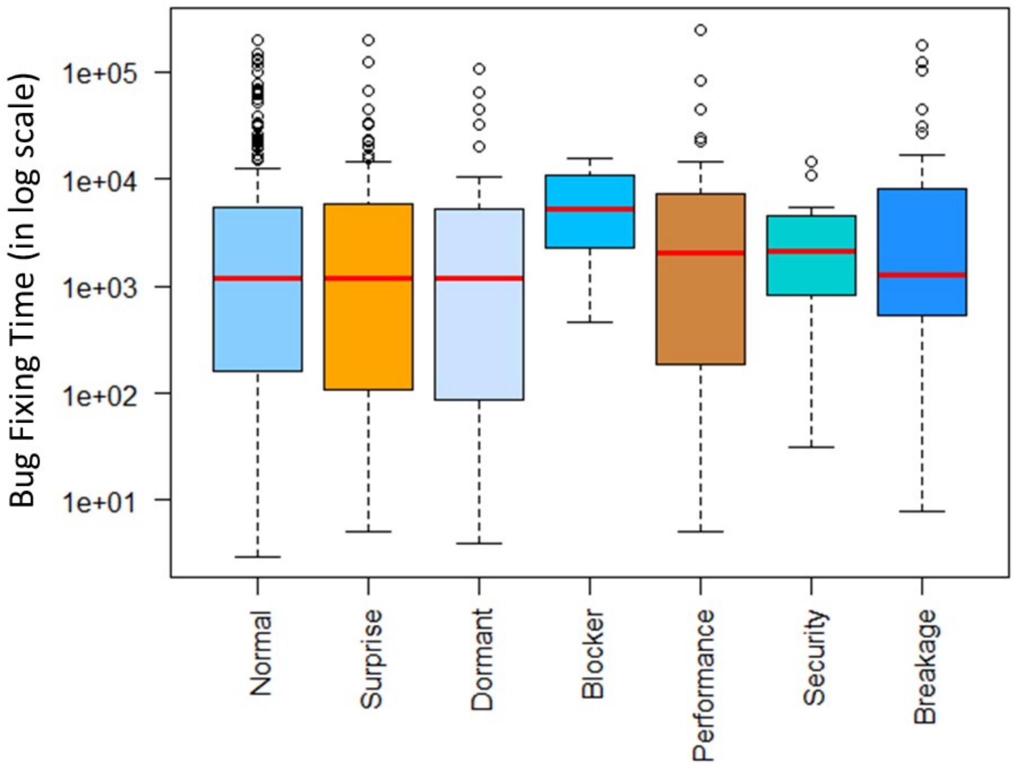


Figure 8.4: Distribution bug fixing time normal vs high-impact

Appendix C

Predicting Key Features

Java programming with Weka-api

To build classification models we use Java programming with Weka-api. We include here the source codes that we used to build classification model.

```java

package smote;
import java.io.File;
import java.io.FileWriter;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Random;

import weka.classifiers.Classifier;
import weka.classifiers.bayes.NaiveBayes;
import weka.classifiers.bayes.NaiveBayesMultinomial;
import weka.classifiers.functions.SMO;
import weka.classifiers.lazy.IBk;
import weka.core.Instance;
import weka.core.Instances;
import weka.core.converters.ConverterUtils.DataSource;
import weka.filters.Filter;
import weka.filters.supervised.instance.SMOTE;
import weka.filters.unsupervised.attribute.StringToWordVector;
//import weka.core.Stopwords;

public class Fp_Smote_New {

  public static void main(String agrs[]) throws Exception{
```

```

 File folder = new File("/Research/DoctoralReseacher/IEICE/Dataset/2NDReview/");
 File[] listOfFiles = folder.listFiles();

     for (int i11 = 0; i11 < listOfFiles.length; i11++) {
       if (listOfFiles[i11].isFile()) {
         if (listOfFiles[i11].getName().endsWith(".arff")){
           String fileName=listOfFiles[i11].getName();
           String[] parFileName=fileName.split(".arff");
           String[] parFileName1=parFileName[0].split("_");

         // System.out.println("File " + parFileName1[0]+parFileName1[1]+parFileName1[2]);
           String fileRootPath = "/Research/DoctoralReseacher/IEICE/Dataset/2NDReview/"+fileName;
           //String fileRootPath = "/Research/DoctoralReseacher/HIB_Classification/JCST/JCST/NewDatasets/"+bugType+"/"+projectName+".arff";
           Instances rawData = DataSource.read(fileRootPath);
           //Stopwords sw=new Stopwords();
           StringToWordVector filter = new StringToWordVector(10000);
           filter.setInputFormat(rawData);
           String[] options = { "-W", "10000", "-L", "-M", "2",
               "-stemmer", "weka.core.stemmers.IteratedLovinsStemmer",
               "-stopwords-handler", "weka.core.stopwords.Rainbow",
               "-tokenizer", "weka.core.tokenizers.AlphabeticTokenizer"
               };

           //sw.add("can");
           filter.setOptions(options);
           filter.setIDFTransform(true);
           filter.setStopwords(new File("/Research/DoctoralReseacher/IEICE/Dataset/stopwords.txt"));
           Instances data = Filter.useFilter(rawData,filter);
           //System.out.println(data);
```

```


            data.setClassIndex(0);

            int numRuns = 10;
            double[] recall=new double[numRuns];
              double[] precision=new double[numRuns];
              double[] fmeasure=new double[numRuns];

            double tp, fp, fn, tn;
            //String classifierName[] = { "KNN"}; //{ "NBM", "KNN", "NB", "SVM"};
            //String classifierName[] = { "KNN","NB"};
            String classifierName[] = { "NBM", "KNN","NB", "SVM", "ZR"}; //{ "BG"};
            //String classifierName[] = { "NBM", "KNN","NB", "SVM","BG"};
            //String classifierName[] = { "BG"}; //{ "BG"};
            Date dNow = new Date( );
              SimpleDateFormat ft =new SimpleDateFormat ("YYMMdd'-'hhmm");
                String cdt=ft.format(dNow);

            File file=new File("/Research/DoctoralReseacher/IEICE/Outputset/2NDReview/"+"WithZeroSmote_"+parFileName1[0]+"_"+parFileName1[3]+"_.txt");
            //File file=new File("/Research/DoctoralReseacher/HIB_Classification/JCST/JCST/NewOutput/"+bugType+"/"+projectName+"_"+classifierName[0]+"_smote.txt");
            FileWriter fw=new FileWriter(file);
            fw.write("Classifier"+"\t"+"----Precision------"+"\t"+"-----Recall------"+"\t"+"-----F-measure------"+"\r\n");
            for(String name:classifierName){
                double totalPrecision,totalRecall,totalFmeasure;
              totalPrecision=totalRecall=totalFmeasure=0;
              double avgPrecision, avgRecall, avgFmeasure;
              avgPrecision=avgRecall=avgFmeasure=0;

              for(int run = 0; run < numRuns; run++){
              System.out.println(name);
```

```
        Classifier classifier = null;

        if (name.equals("NBM"))
          classifier = new NaiveBayesMultinomial();

        if (name.equals("ZR"))
          classifier = new ZeroR();

        if (name.equals("NB"))
          classifier = new NaiveBayes();

        if (name.equals("KNN"))
          classifier = new IBk();

        if (name.equals("SVM"))
          classifier = new SMO();

        if (name.equals("RF"))
          classifier = new RandomForest();

        //if (name.equals("BG"))
        //  classifier = new Bagging();

        if (name.equals("BG"))
        {
          Bagging bagger = new Bagging();
          bagger.setClassifier(new RandomForest());
          //bagger.setSeed(2);
          classifier=bagger;

            //classifier.setClassifer(new RandomTree())
        }

        if (name.equals("ST"))
          classifier = new Stacking();

        if (name.equals("STC"))
          classifier = new StackingC();

        if (name.equals("VT"))
```

```
                classifier = new Vote();

              if (name.equals("ADB"))
              {
                AdaBoostM1 m1 = new AdaBoostM1();
                  m1.setClassifier(new DecisionStump());//needs one base-classifier
                  //m1.setNumIterations(10);
                  //m1.buildClassifier(trainingData);
                  classifier = m1;
              }

              //cross validation
              int folds = 10;
              Random random = new Random(1);
              data.randomize(random);
              data.stratify(folds);


              tp = fp = fn = tn = 0;
              for (int i = 0; i < folds; i++) {

                Instances trains = data.trainCV(folds, i,random);
                Instances tests = data.testCV(folds, i);

                //smote
                SMOTE smote=new SMOTE();
                smote.setInputFormat(trains);
                Instances smoteTrains = Filter.useFilter(trains, smote);

                classifier.buildClassifier(smoteTrains);
                for (int j = 0; j < tests.numInstances(); j++) {

                  Instance instance = tests.instance(j);

                  double classValue = instance.classValue();

                  double result = classifier.classifyInstance(instance);
```

```

                    if (result == 0.0 && classValue == 0.0) {
                      tp++;
                    } else if (result == 0.0 && classValue == 1.0) {
                      fp++;
                    } else if (result == 1.0 && classValue == 0.0) {
                      fn++;
                    } else if (result == 1.0 && classValue == 1.0) {
                      tn++;
                    }
                  }
                }

                if (tn + fn > 0)
                  precision[run] = tn / (tn + fn);
                if (tn + fp > 0)
                  recall[run] = tn / (tn + fp);
                if (precision[run] + recall[run] > 0)
                  fmeasure[run] = 2 * precision[run] * recall[run] /
     (precision[run] + recall[run]);
                System.out.println("The "+(run+1)+"-th run");
                System.out.println("Precision: " + precision[run]);
                System.out.println("Recall: " + recall[run]);
                System.out.println("Fmeasure: " + fmeasure[run]);
                totalPrecision+=precision[run];
                  totalRecall+=recall[run];
                  totalFmeasure+=fmeasure[run];

              }
                  avgPrecision=totalPrecision/numRuns;
                  avgRecall=totalRecall/numRuns;
                  avgFmeasure=totalFmeasure/numRuns;
                  System.out.println("avgPrecision: " + avgPrecision
    );
                System.out.println("avgRecall: " + avgRecall);
                System.out.println("avgFmeasure: " + avgFmeasure);
                  fw.write(name+"        "+"\t"+avgPrecision+"\t"+
    avgRecall+"\t"+avgFmeasure+"\r\n");
              }
              fw.close();
```

```


                    }
                }

            }

    }
}
```

R Programming

We use the following R code to conduct significance test between two classifications.

```
C1_name='NB'
C2_name='SVM'
x1=subset(thunderbird_EB, select=c(F_measure), subset=c(trimws(
    Classifier, which=c("both"))==C1_name))
x2=subset(thunderbird_EB, select=c(F_measure), subset=c(trimws(
    Classifier, which=c("both"))==C2_name))
wilcox.test(x1[[1]],x2[[1]], alternative = c("two.sided"))
```