AUTOMATIC BUG TRIAGING TECHNIQUES USING MACHINE LEARNING AND STACK TRACES

KOROSH KOOCHEKIAN SABOR

A THESIS
IN
THE DEPARTMENT
OF
ELECTRICAL AND COMPUTER ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY (ELECTRICAL AND COMPUTER ENGINEERING) AT

CONCORDIA UNIVERSITY
MONTREAL, QUEBEC, CANADA

August 2019

© KOROSH KOOCHEKIAN SABOR, 2019

CONCORDIA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Korosh Koochekian Sabor

Entitled: Automatic bug triaging techniques using machine learning and stack traces

and submitted in partial fulfillment of the requirements for the degree of

Doctor Of Philosophy (Electrical and Computer Engineering)

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

______________________________ Dr. Luis Amador	Chair
______________________________ Dr. Ali Ouni	External Examiner
______________________________ Dr. Nikos Tsantalis	External to Program
______________________________ Dr. Anjali Agarwal	Examiner
______________________________ Dr. Nawwaf Kharma	Examiner
______________________________ Dr. Abdelwahab Hamou-Lhadj	Thesis Supervisor

Approved by ___________________________________
Dr. Rastko R. Selmic, Graduate Program Director

September 27, 2019 ______________________________
Dr. Amir Asif, Dean
Gina Cody School of Engineering & Computer Science

Abstract

Automatic Bug Triaging Techniques Using Machine Learning and Stack Traces

Korosh Koochekian Sabor, Ph.D.

Concordia University, 2019

When a software system crashes, users have the option to report the crash using automated bug tracking systems. These tools capture software crash and failure data (e.g., stack traces, memory dumps, etc.) from end-users. These data are sent in the form of bug (crash) reports to the software development teams to uncover the causes of the crash and provide adequate fixes. The reports are first assessed (usually in a semi-automatic way) by a group of software analysts, known as triagers. Triagers assign priority to the bugs and redirect them to the software development teams in order to provide fixes.

The triaging process, however, is usually very challenging. The problem is that many of these reports are caused by similar faults. Studies have shown that one way to improve the bug triaging process is to detect automatically duplicate (or similar) reports. This way, triagers would not need to spend time on reports caused by faults that have already been handled. Another issue is related to the prioritization of bug reports. Triagers often rely on the information provided by the customers (the report submitters) to prioritize bug reports. However, this task can be quite tedious and requires tool support. Next, triagers route the bug report to the responsible development team based on the subsystem, which caused the crash. Since having knowledge of all the subsystems of an ever-evolving industrial system is impractical, having a tool to automatically identify defective subsystems can significantly reduce the manual bug triaging effort.

The main goal of this research is to investigate techniques and tools to help triagers process bug reports. We start by studying the effect of the presence of stack traces in analyzing bug reports. Next, we present a framework to help triagers in each step of the bug triaging process. We propose a new and scalable method to automatically detect duplicate bug reports using stack traces and bug report categorical features. We then propose a novel approach for predicting bug severity using stack traces and categorical features, and finally, we discuss a new method for predicting faulty product and component fields of bug reports.

We evaluate the effectiveness of our techniques using bug reports from two large open-source systems. Our results show that stack traces and machine learning methods can be used to automate the bug triaging process, and hence increase the productivity of bug triagers, while reducing costs and efforts associated with manual triaging of bug reports.

Acknowledgments

First and foremost, I want to express my profound and sincere gratitude to my respectful supervisor, Dr. Abdelwahab Hamou-Lhadj. Thanks for your trust in me, and thank you for proactively providing me all the academic, mental, and financial support I needed throughout the program. I would like to express my heartfelt gratitude for your mentorship, which helped me grow both professionally, personally and helped me to further develop my future career path.

My deep gratitude for all the support I received from my PhD committee, Dr. Nikolaos Tsantalis, Dr. Nawwaf Kharma, Dr. Ali Ouni and Dr. Anjali Agarwal. Their suggestions significantly helped me in further improving the direction of my research.

I would like to extend my thanks to Alf Larsson from Ericsson, Sweden, whose feedback and comments throughout this work have been very valuable. I would also like to thank Ericsson, MITACS, NSERC, and the Gina School of Engineering and Computer Science at Concordia University for their financial support.

Many thanks to all my friends at Concordia University. I was thrilled to have been surrounded by such wonderful people. Special thanks to Mohammad Reza Rejali and Amir Bahador Gahroosi for being such wonderful friends, I would be indebted for what you did for me during our collaboration and friendship in Concordia. Thanks to all my friends outside of Concordia, including Mojataba Khomami Abadi and Parham Darbandi, for all their friendship and support.

Words cannot express my gratitude and gratefulness enough to my beloved family members. My deepest appreciation for my Parents Anoushe and Saeedeh, for inspiring me to start the program and giving me heartwarming all the way through my program, my beloved sister, Camelia, who always supported my decisions from thousands miles away, and very special thanks to my wife, Camelia, I am very blessed to have you in my life, thanks for your extraordinary patience, care, kindness, love and all the sacrifices you made throughout the program. I could not have accomplished this work without your unconditional support.

Table of Contents

1 Introduction 1
1.1. Terminology 4
1.2. Research Hypothesis 4
1.3. Thesis Contributions 5
1.3.1. Chapter 2: Background and Related Work 6
1.3.2. Chapter 3: Data Preparation 6
1.3.3. Chapter 4: An Empirical Study on the Effectiveness of Stack traces 7
1.3.4. Chapter 5: Detecting Duplicate Bug Reports 7
1.3.5. Chapter 6: Predicting Severity of Bugs 7
1.3.6. Chapter 7: Predicting Faulty Product and Component Fields of Bug Reports . 7
1.3.7. Chapter 8: Conclusion and future work 7
2 Background and Related Work 8
2.1. Background 8
2.1.1. Bug Report Description 8
2.1.2. Stack Trace 8
2.1.3. Categorical Information 9
2.2. Related Work 10
2.2.1. Usefulness of Stack Traces 10
2.2.2. Detection of Duplicate Bug Reports 11
2.2.3. Predicting Bug Severity 18
2.2.4. Bug Report Faulty Product and Component Field Prediction 24
3 Data Preparation 28
3.1. Eclipse Dataset 28
3.2. Gnome Dataset 29
4 Empirical Study on the Effectiveness of Presence of Stack Traces 31
4.1. Dataset Setup 31
4.2. Statistical Analysis 32
4.3. Experiments 32
4.4. Study Result 36
4.5. Discussion 43
5 Detecting Duplicate Bug Reports 44
5.1. Preliminaries 45
5.2. Proposed Approach 48
5.3. Evaluation 51
5.3.1. The Dataset 52
5.3.2. Dataset Analysis 53
5.3.3. Evaluation Measure 55
5.4. Results and Discussion 56

5.4.1. Comparison of DURFEX to the of Function Calls 56
5.4.2. Comparison of DURFEX to the of Function Calls with a Time Window 58
5.4.3. Comparison of Processing Time Using Functions and Packages 60
5.5. Threats to Validity 62
5.6. Conclusion 62
6 Bug Severity Prediction 63
6.1. Bug Report Features 64
6.2. Levels of Bug Severity 65
6.3. The Proposed Approach 67
6.3.1. Predicting the Bug Severity Using Stack Traces and Categorical Features 67
6.3.2. Predicting Severity Using Stack Traces 69
6.3.3. Bug Features Extraction (Stack Traces and Categorical Features) 69
6.3.4. Stack Trace Similarity 70
6.3.5. KNN Classifier for Severity Classification 70
6.3.6. Overall Approach 72
6.4. Evaluation 75
6.4.1. Experimental Setup 76
6.4.2. Cost-sensitive K Nearest Neighbour 79
6.4.3. Predicting Severity of Bugs Using Description 83
6.4.4. Predicting Severity of Bugs Using a Random Classifier 84
6.4.5. Severity Prediction Approaches Setup 85
6.4.6. Evaluation Metrics 85
6.4.7. Evaluation Results 86
6.5. Threats to Validity 104
6.5.1. Threats to External Validity 104
6.5.2. Threats to Internal Validity 107
6.6. Conclusion 107
7 Automatic Prediction of Bug Report Faulty product and component fields 109
7.1. Bug Report Features 111
7.2. The Proposed Approach 111
7.2.1. Predicting Bug Report Faulty Product and Component Fields 111
7.2.2. Bug Features Extraction 113
7.2.3. KNN Classifier for Faulty Component and Product Classification 114
7.2.4. Overall Approach 116
7.3. Evaluation 118
7.3.1. Experimental Setup 118
7.3.2. Predicting Faulty Product and Component Fields Using Description 121
7.3.3. Predicting Faulty Product and Component Fields Random Approach 121
7.3.4. Faulty Product and Component Prediction Approaches Setup 121
7.3.5. Evaluation Metrics 122
7.3.6. Evaluation Results 123
7.3.7. Implication and limitations 135

7.4. Threats to Validity 136
7.4.1. Threats to External Validity 136
7.4.2. Threats to Internal Validity 136
7.4.3. Threats to Construct Validity 137
7.5. Conclusions and Future Work 137
8 **Conclusion and future work** **139**
8.1. Thesis Findings 140
8.2. Future Work 142
8.2.1. Limitations 142
8.2.2. Future Research Opportunities 143
References **145**

List of Figures

Figure 1. Bug Handling Process 3
Figure 2. An overview of the automatic bug triaging using stack traces 5
Figure 3. The stack trace for bug report 38601 from Eclipse bug repository 9
Figure 4. creating a new bug report in Eclipse Bugzilla 9
Figure 5. Regular expression for extracting stack traces from bug reports Eclipse 28
Figure 6. Regular expression for extracting stack traces from bug report Gnome 30
Figure 7. Data collection and analysis 33
Figure 8. Severe and non-severe bugs percentage based on stack trace existence Eclipse .38
Figure 9. Severe and non-severe bugs percentage based on stack trace existence Gnome.38
Figure 10. Percentage of bug reports based on existence of stack trace in Eclipse 40
Figure 11. Percentage of bug reports based on existence of stack trace in Gnome 40
Figure 12. Percentage of bug reports based on existence of stack trace in Eclipse. 41
Figure 13. Percentage of bug reports based on existence of stack trace in Gnome. 41
Figure 14. Variable length N-gram 46
Figure 15. Training Dataset 49
Figure 16. Optimization using gradient descent..50
Figure 17. Proposed approach (DURFEX) 51
Figure 18. The number of duplicate reports in all duplicate bugs groups in Eclipse 53
Figure 19. Cumulative number of distinct functions and packages in Eclipse 54
Figure 20. Days between the first and last bug report in the duplicate group in Eclipse 55
Figure 21. Recall rate of DURFEX, different N-grams unique functions in Eclipse 57
Figure 22. Recall rate of DURFEX, different N-grams distinct functions in 100 day Eclipse ..59
Figure 23. Comparison of processing time of in Eclipse dataset 61
Figure 24. Training Dataset 68
Figure 25. Overall Approach..73
Figure 26. Example of online severity prediction approach 74
Figure 27. The bug handling process...77
Figure 28. Distribution of the severity labels in Eclipse and Gnome datasets.......................79
Figure 29. Confusion matrix 81
Figure 30. Cost of predicting each severity label...82
Figure 31. Updated testing phase using cost sensitive k nearest neighbour 83
Figure 32. F-measure of predicting severity by varying list size in Eclipse Critical Severity ..87
Figure 33. F-measure of predicting severity by varying list size in Eclipse Blocker Severity..88
Figure 34. F-measure of predicting severity by varying list size in Eclipse Major Severity 89
Figure 35. F-measure of predicting severity by varying list size in Eclipse Minor Severity....90
Figure 36. F-measure of predicting severity by varying list size in Eclipse Trivial Severity 91
Figure 37. F-measure of predicting severity by varying list size in Gnome Critical Severity..92
Figure 38. F-measure of predicting severity by varying list size in Gnome Blocker Severity.93
Figure 39. F-measure of predicting severity by varying list size in Gnome Major Severity ...94
Figure 40. F-measure of predicting severity by varying list size in Gnome Minor Severity ...95

Figure 41. F-measure of predicting severity by varying list size in Gnome Trivial Severity ...96
Figure 42. Eclipse bug report #215679 history information ...110
Figure 43. Training dataset ...113
Figure 44. Overall approach ...117

List of Tables

Table 1. Eclipse dataset charchteristics 29
Table 2. Gnome dataset charchteristics 30
Table 3. characteristics of the dataset 52
Table 4. DURFEX recall rate on the Eclipse dataset 57
Table 5. DURFEX Mean reciprocal rank on the Eclipse dataset 58
Table 6. DURFEX results on the bug reports of the 100-day period on Eclipse 59
Table 7. DURFEX Mean reciprocal rank of the 100-day period on Eclipse 60
Table 8. Comparing the average execution time of DURFEX 61
Table 9. Characteristics of the datasets 78
Table 10. Severity prediction accuracy (Eclipse Critical severity) 87
Table 11. Severity prediction accuracy (Eclipse Blocker severity) 88
Table 12. Severity prediction accuracy (Eclipse Major severity) 89
Table 13. Severity prediction accuracy (Eclipse Minor severity) 90
Table 14. Severity prediction accuracy (Eclipse Trivial severity) 91
Table 15. Severity prediction accuracy (Gnome Critical severity) 92
Table 16. Severity prediction accuracy (Gnome Blocker severity) 93
Table 17. Severity prediction accuracy (Gnome Major severity) 94
Table 18. Severity prediction accuracy (Gnome Minor severity) 95
Table 19. Severity prediction accuracy (Gnome Trivial severity) 96
Table 20. Statistical tests of stack traces and categorical features approach for Eclipse 100
Table 21. Statistical tests of stack traces and categorical features approach for Gnome 100
Table 22. Statistical tests of stack traces approach for Eclipse 100
Table 23. Statistical tests of stack traces approach for Gnome 101
Table 24. Eclipse Bug Report #296383 103
Table 25. Eclipse Bug Report #313534 103
Table 26. Gnome Bug Report#273727 105
Table 27. Gnome Bug Report #532680 106
Table 28. Products and components in Eclipse dataset 119
Table 29. Products and components in Gnome dataset 120
Table 30. Characteristics of the datasets 121
Table 31. Product prediction accuracy for Eclipse. 126
Table 32. Product prediction accuracy for Gnome 126
Table 33. Component prediction accuracy for Eclipse 126
Table 34. Components prediction accuracy for Gnome 126
Table 35. Component prediction accuracy (Eclipse Equinox Product) 127
Table 36. Component prediction accuracy (Eclipse PDE Product) 127
Table 37. Component prediction accuracy (Eclipse E4 Product) 127
Table 38. Component prediction accuracy (Eclipse JDT Product) 127
Table 39. Component prediction accuracy (Eclipse Platform Product) 128
Table 40. Component prediction accuracy (Gnome Deprecated Product) 128

Table 41. Component prediction accuracy (Gnome Other Product)....129
Table 42. Component prediction accuracy (Gnome infrastructure Product)....129
Table 43. Component prediction accuracy (Gnome Binding Product)....129
Table 44. Component prediction accuracy (Gnome Platform Product)....129
Table 45. Component prediction accuracy (Gnome Core Product)....130
Table 46. Component prediction accuracy (Gnome Applications Product)....130
Table 47. Eclipse Bug Report #213234....131
Table 48. Eclipse Bug Report#213234....132
Table 49. Eclipse Bug Report#192746....133
Table 50. bug report#408425....134
Table 51. Bug report#404634....134

List of publications resulting from this thesis is as follows:

- Korosh Koochekian Sabor, Abdelwahab Hamou-Lhadj and Alf Larsson, “DURFEX: A Feature Extraction Technique for Efficient Detection of Duplicate Bug Reports,” In Proceedings of the IEEE International Conference on Software Quality, Reliability and Security (QRS), 2017, pp 240-250.
- Korosh Koochekian Sabor, Mohammad Hamdaqa, and Abdelwahab Hamou-Lhadj, “Automatic prediction of the severity of bugs using stack traces,” In Proceedings of the 26th IBM Annual International Conference on Computer Science and Software Engineering (CASCON '16), 2016, pp 96-105.
- Korosh Koochekian Sabor, Mohammad Hamdaqa, and Abdelwahab Hamou-Lhadj, “Automatic prediction of the severity of bugs using stack traces and categorical features,” Elsevier Journal of Information and Software Technology (IST), 2019.
- Korosh Koochekian Sabor, Abdelwahab Hamou-Lhadj, Jameleddine Hassine, Abdelaziz Trabelsi, “Predicting bug report fields using stack traces and categorical attributes,” In Proceedings of the 28th IBM Annual International Conference on Computer Science and Software Engineering (CASCON '19), 2019, pp 224-233.
- Korosh Koochekian Sabor, Mathieu Nayrolles, Abdelaziz Trabelsi, Abdelwahab Hamou-Lhadj, “An Approach for Predicting Bug Report Fields Using a Neural Network Learning Model,” In Proceedings of the IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), 2018.
- Abdo Maiga, Abdelwahab Hamou-Lhadj, Mathieu Nayrolles, Korosh Koochekian Sabor and Alf Larsson, "An empirical study on the handling of crash reports in a large software

company: An experience report," In Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME), 2015, pp 342-351.

Chapter 1

Introduction

Software systems play a critical role in almost every industry sector, including Telecom, finance, public safety, education, etc. Failure of these systems may have important economic impacts. For example, a study showed that software failures cost the U.S. economy $59 billion every year [N2], which amounts to 0.6% of the gross domestic product of the United States [N2].

The problem is that it is almost impossible to guarantee the absence of bugs in released systems. This is due to many factors. First, exhaustive testing is known to be impossible. Second, the pressure to release new products on the market as quickly as possible often comes at the price of quality. In addition, continuous maintenance activities are prone to the introduction of new bugs in the system. As a result, many software systems continue to crash during operation.

When a system crashes, users have the option to report the crash using automated bug tracking systems such as the Windows Error Reporting tool[1], the Mozilla crash reporting system[2], and Ubuntu's Apport crash reporting tool[3]. These tools capture software crash and failure data (e.g., stack traces, memory dumps, etc.) from end-users.

This data is sent in the form of bug (crash) reports to the software development teams to uncover the causes of the crash and provide adequate fixes. The reports are first assessed (usually in a semi-automatic way) by triagers. Triagers route bug reports to the software development teams in order to provide fixes. The manual triaging process, however, is usually very challenging since there are just many bug reports that are submitted every day. For example, on April 24, 2002, Eclipse users have submitted over 107 bug

[1] http://msdn.microsoft.com/en-us/library/windows/hardware/gg487440.aspx
[2] http://crash-stats.mozilla.com
[3] https://wiki.ubuntu.com/Apport

reports. In the Gnome bug tracking system, over 108 bug reports were submitted by users on June 22, 2000.

When a bug is reported to the bug tracking system, it goes through a triaging process. First, triagers needs to examine if the incoming bug report is duplicate of previous bug reports, i.e., caused by a similar fault. Studies have shown that one way to improve the bug triaging process is to automatically detect duplicate bug reports [LM13, BPZK08a]. If a bug report is deemed to be a duplicate of an existing one, then triagers can mark it as duplicate and stop the process, saving time and effort.

Next, since the available resources for the development and maintenance of software systems are limited, prioritizing bug reports helps triagers identify the bugs that need to be fixed first based on the availability of resources. Triagers often rely on the information provided by the users (the report submitters) to prioritize the bug reports. However, since users are usually not familiar with the software system, the information provided by them could be inaccurate. Studies have shown that the severity of the bugs can be predicted automatically [LDGG10, LDSV11], which helps triagers to prioritize the bugs and hence speed up the bug handling process.

Triagers route bug reports that require fixes to the development team. Since each development team is usually focused on the development of a specific product and component of the software system, triagers use the faulty product and component fields of a bug report to identify the development team that can provide the fix. Because users (bug report submitters) are usually not familiar with the architecture of the software system, they may choose incorrect faulty products and components when reporting bugs, causing bug reports to be routed to the incorrect development team, which often delays the fixing time of the bugs. Triagers need to adjust the faulty product and component fields of bug reports based on other information provided by the user to avoid assignment of bug reports to incorrect teams. Based on the adjusted faulty product and component fields, they can then decide which development team has the expertise to deal with the bug report. Studies (e.g., [S12]) have shown that automatic prediction of

the correct component and product fields of bug reports can improve the bug handling process significantly by routing the bug report to the correct development team.

Figure 1 shows an overview of the bug triaging framework that illustrates the three main activities of the triaging teams, namely, detection of duplicate bug reports, determination of bug report severity, and routing of bug reports to developers.

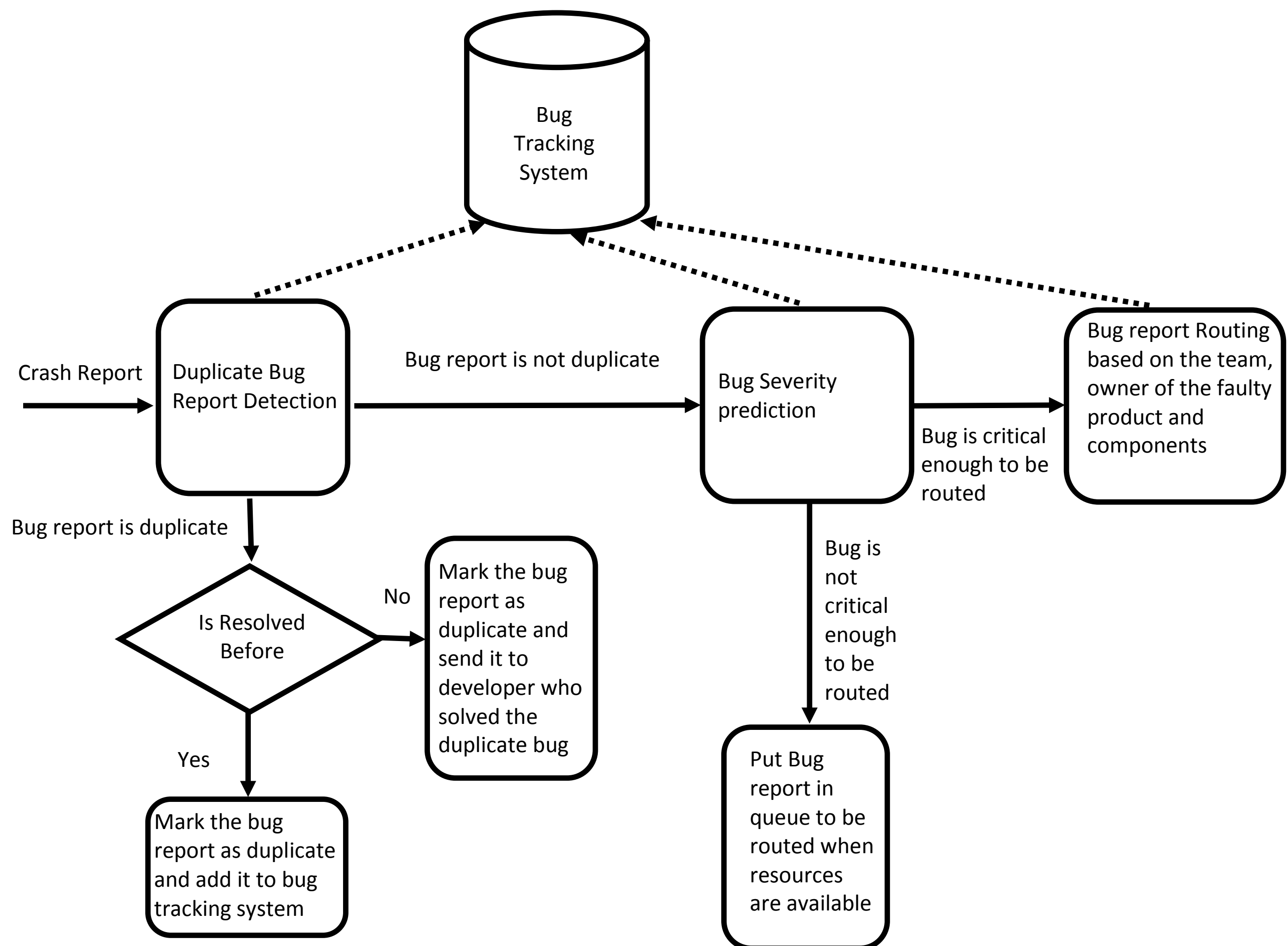

Figure 1 Bug Handling Process

1.1. Terminology

In this Section, we review the terminologies used to characterize bugs, faults, errors, and crashes.

Software Fault (Bug, Defect): A software fault is a static defect caused by a human error. A fault can be considered as a design or a programming mistake [AO08].

Software Error: A software error is an incorrect internal state of software caused by one or more software faults [AO08]. This incorrect internal state might or might not manifest itself as an external behaviour.

Software Failure: Many faults may stay dormant and never manifest themselves. Software failure is defined as any incorrect external behaviour of the software system, which is in contrast to the functional or non-functional requirements of the software. Failure is an external manifestation of an error caused by some faults [AO08].

Software Crash: Failure is defined as an unexpected output according to the software requirement. One of the unexpected outputs can be that the software stops working because of some faults, which is defined as a software crash.

Bug report: A bug report is a report created by a user when facing a crash in the software system. A bug report can also be submitted automatically by a crash reporting tool. We use bug report and crash report terms interchangeably in this thesis.

Duplicate Bug reports: Duplicate bug reports are bug reports that have the same underlying fault [RAN07].

1.2. Research Hypothesis

This thesis aims to provide an efficient and accurate automatic bug triaging system, which can be implemented as part of a bug tracking framework, to reduce the overhead of manual bug triaging process significantly. Our system offers the following capabilities: duplicate bug reports detection, automatic prediction of severity of bugs, and automatic prediction of faulty product and component fields of bug

reports. To this end, we leverage stack traces and categorical features of bug reports, and machine learning algorithms.

The thesis statement is:

Stack traces and categorical features of bug reports contain valuable information that can be used to automatically detect duplicate bug reports, predict the severity of bugs, and predict faulty product and component fields of bug reports.

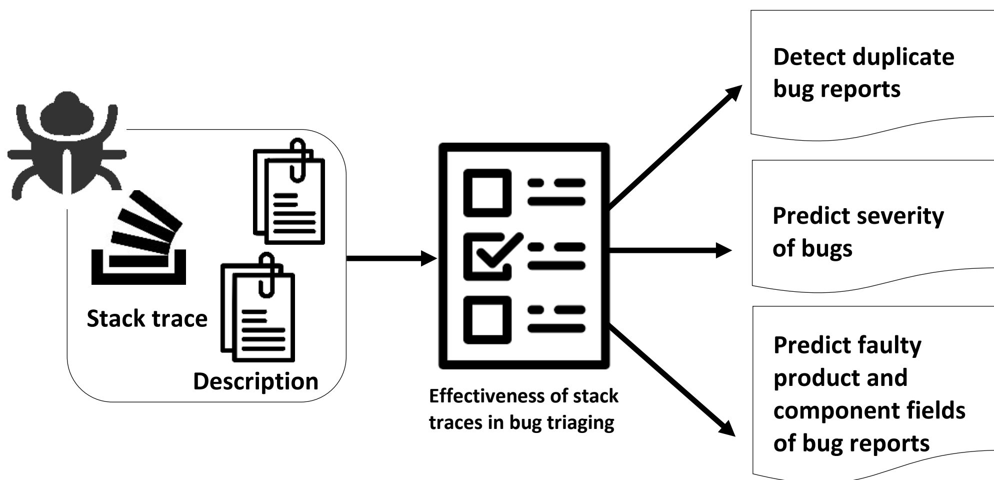

Figure 2 An overview of the automatic bug triaging using stack traces

1.3. Thesis Contributions

An overview of the contributions of this thesis is shown in Figure 2.

- We study the usefulness of stack traces in the bug triaging process. This study sheds light on how the presence of stack traces improves the bug triaging process.
- We propose an approach for detecting duplicate bug reports. The approach leverages the concept of trace abstraction to facilitate the detection of duplicate bug reports.
- We propose an approach for predicting the severity of bugs using stack traces and the combination of stack traces and categorical features.
- We propose an approach for predicting product and component fields of bug reports using stack traces and categorical features.

1.3.1. Chapter 2: Background and Related Work

In Chapter 2, we present background information on the content of bug reports. Next, we focus on related studies in the bug triaging. We group these studies into four pillars:

- **Empirical studies on bug report information**. Prior studies in this aspect explore the factors that impact the quality of bug reports. These studies quantify the importance of each type of information provided in the bug reports by identifying the most important information and providing guidelines on how providing those sets of information could facilitate the bug triaging process.

- **Detecting duplicate bug reports.** Duplicate bug report detection is a very important step in bug triaging. Duplicate bug reports, if go undetected, impose huge overhead on the bug triaging process. However, if detected properly, duplicate bug reports can provide additional information to developers to fix the bugs. Studies on duplicate bug report detection mainly focus on providing a set of potential duplicate bug reports for each incoming bug report [LM13].

- **Predicting the severity of bugs.** Users chose a severity for each bug report submitted to the bug tracking system. Severity prediction studies aim to automatically predict the severity of bugs to minimize misinterpretation of bug report severity labels reported by users and help triagers to optimize the bug resolution process by accurate prioritization of bug reports.

- **Predicting faulty product and component fields of bug reports.** Users chose the faulty product and component when reporting bug reports. Many studies focused on proposing techniques to automatically predict the faulty product and component fields of bug reports. These studies use the history of past bug reports to predict potential faulty product and component fields of the new bug reports.

1.3.2. Chapter 3: Data Preparation

In Chapter 3, we present two publicly available datasets that are used to evaluate approaches proposed in this thesis. Both datasets were subject of mining challenges in Mining Software Repositories conference.

1.3.3. Chapter 4: An Empirical Study on the Effectiveness of Stack traces

Stack traces are shown to be one of the most reliable sources of information used by developers to fix bugs [BJSWPZ08]. In chapter 4, we empirically study the effect of the presence of stack traces in the bug triaging process. The outcome of the study shows that stack traces improve the bug triaging process. The study shows that there is a strong potential to use the content of stack traces to automate the bug triaging process.

1.3.4. Chapter 5: Detecting Duplicate Bug Reports

In this chapter, we propose a feature reduction technique based on the trace abstraction concept, which reduces the processing overhead of detecting duplicate bug reports, while providing good accuracy.

1.3.5. Chapter 6: Predicting Severity of Bugs

Various techniques have been introduced to predict the severity of bugs using bug report descriptions. Following the promising result of stack traces in the improvement of the bug triaging process, we propose an approach, which predicts the severity of bugs using stack traces. We also show that using categorical features in addition to stack traces further improves the severity prediction accuracy.

1.3.6. Chapter 7: Predicting Faulty Product and Component Fields of Bug Reports

Various methods in the literature have been proposed to predict faulty product and component fields of bug reports using bug report descriptions. In Chapter 7, we focus on using stack traces and categorical features in predicting faulty product and component fields of bug reports. We show that stack traces and categorical features can predict faulty product and component fields with higher accuracy than bug report descriptions.

1.3.7. Chapter 8: Conclusion and future work

In chapter 8, in the first section, we focus on thesis findings. Next, we present the thesis limitations and provide a list of future research opportunities.

Chapter 2
Background and Related Work

In this chapter, we elaborate on the information which the bug report provides to the triagers. Next, we explain related work on the empirical study of the effectiveness of stack traces in bug triaging, duplicate bug report detection, bug severity prediction, and predicting faulty product and component fields of bug reports.

2.1. Background

2.1.1. Bug Report Description

Bug report description is mainly used to explain the defective behaviour of the software system. A good description should provide enough information to guide developers in understanding what the user or tester has observed during the crash. Bug report descriptions are usually verbose. An example of a well-written description is "Download fails on Customer Report page when a user clicks on the download button” meanwhile, an example of an inefficient description is "Download does not work", as it does not provide details about when or how this failure occurs. The quality of a bug report description depends on who is writing this description and hence it is subjective.

2.1.2. Stack Trace

A stack trace is a sequence of function calls that ideally lists all function calls from the moment a software program starts execution to the crash point. In some cases, the stack trace is trimmed to contain a window of functions before the crash. An example of a stack trace is shown in Figure 3. The stack trace in Figure 3 is taken from the Eclipse bug repository. It represents the stack trace of Bug report 38601. The bug was caused by a failure of checking for a null pointer in the search for a method reference.

5- org.eclipse.jdt.internal.corext.util.Strings.convertIntoLines()

4- org.eclipse.jdt.internal.ui.text.java.hover.JavaSourceHover.getHoverInfo()

3- org.eclipse.jdt.internal.ui.text.java.hover.AbstractJavaEditorTextHover.getHoverInfo()

2- org.eclipse.jdt.internal.ui.text.java.hover.JavaEditorTextHoverProxy.getHoverInfo()

1- org.eclipse.jface.text.TextViewerHoverManager.run()

Figure 3 The stack trace for bug report 38601 from Eclipse bug repository

2.1.3. Categorical Information

In addition to the description and stack trace, users must choose other categorical fields, which further help developers to fix the bug. This categorical information includes the product, component, version, operating system, and severity of the bug. Each software application is composed of several products, and each product is composed of several components. Users should choose the product and the component that they think the bug is related to, when reporting a bug. Version indicates the version of the software that the user was using when the crash happened. Operating system shows the name and release number of the operating system in which the software was running on when the crash happened. Also, severity shows how much a bug affects normal execution of program [LDGG10]. Figure 4 shows the information which user needs to provide when reporting a bug.

Product: Platform
Reporter: cyrus1_thegreat@yahoo.com
Component: Ant, Compare, CVS, Debug, Doc, IDE, Incubator
Component Description
Select a component to read its description.
Version: 4.7.3, 4.8, 4.9, 4.10, 4.11
Severity: normal
Hardware: PC
OS: Windows 7
We've made a guess at your operating system and platform. Please check them and make any corrections if necessary.
Summary:
Description: Comment | Preview
Attachment: Add an attachment
Submit Bug

Figure 4 creating a new bug report in Eclipse Bugzilla

2.2. Related Work

2.2.1. Usefulness of Stack Traces

One of the earliest studies that showed the usefulness of stack traces is conducted by Battenberg et al. [BJSWPZ08]. They studied the quality of bug reports and showed that there is an important gap between what users provide as input and what developers need to fix a bug. They argued that because users do not usually have technical knowledge about the system, it is very difficult for them to provide useful information to developers. The authors observed that developers consider stack traces to be one of the most useful information to fix a bug.

Schroter et al. [SBP10] studied the use of stack traces by developers in Eclipse projects. More precisely, they answered four research questions: 1. Are bugs fixed in the methods in stack traces? 2. How far down the stack traces are bugs fixed? 3. Are two stack traces better than one? 4. Do stack traces help speeding up debugging? They used InfoZilla [SBP10] to extract stack traces from Eclipse bug reports for the period of 2001 to 2006. They also mined the Eclipse version repository to extract the relation between bugs and the source code fixes. They showed that 60% of bug reports with stack trace were fixed in one of the stack trace methods. They concluded that developers favour stack traces since stack traces can help them find the fix location easier. They studied the number of methods in stack traces and observed that it ranges from 1 to 1024 methods with a median of 25 methods. They showed that 40% of the bugs are fixed by a patch in the first method of stack traces, 80% of the bugs are fixed by a patch in one of the first six methods of stack traces, and 90% of the bugs are fixed by a patch in one of the first ten methods of stack traces. They also showed that in the presence of multiple stack traces, 70% of the bugs are fixed in one of the methods in the first stack trace, and 90% of the bugs are fixed using one of the methods in the first three stack traces. They also showed that bug reports with stack traces are fixed sooner than bug reports without stack traces. They explained that they hope their study emphasizes the importance of stack traces and encourages users to provide stack traces in bug reports. They also suggested bug tracking tool developers provide more support for capturing stack traces in their tools.

Another study was conducted by Krikke [K12], where the author studied the association between the severity field of Eclipse bug reports and the presence of stack traces. He also studied the association between the time to fix a bug and the presence of stack traces in bug reports. He showed that there is a strong statistical association between severity and the presence of stack traces in the bug reports. He showed that Eclipse bug reports, which have stack traces, are fixed sooner [K12].

The effectiveness of four quality indicators for predicting the severity of bugs is studied by Yang et al. [YCKY14]. They categorized Blocker, Major and Critical bug severities as severe, and Minor and Trivial bug severities as non-severe. They considered stack traces, bug report description length, attachment and step to reproduce as four quality indicators of bug reports in their study. They used the Eclipse dataset to apply their experiments. In the first round of experiments, they only used the presence of those quality indicators, whereas in the second round of experiments, they considered the quantitative value of those indicators too. The quantitate value of each quality indicator is calculated differently. For stack trace, they used the number of functions. For attachment, they considered the number of attachments. For report length, they used the quantitative value of report length, and for step to reproduce, they used the number of steps. They concluded that among those four quality indicators, the stack trace is the best indicator of bug severity.

Stack traces have also been shown to be useful in other bug processing activities. For example, Nayrolles et al. [NHTL15, NHTL16] showed that stack traces can be used to reproduce a bug in Java system, which in turn can speed up the bug fixing process.

2.2.2. Detection of Duplicate Bug Reports

One of the key tasks of triagers is to determine whether the reported bug has been handled before or not. If it is, then it should be marked as duplicate. While some studies such as the one by Jalbert et al. [JW08] have been conducted on the premise that duplicate bug reports should be ignored during bug triaging, other studies such as the one by Bettenburg et al. [BPZK08a] believe that duplicate

reports provide additional information that can be used by developers to solve the problem faster. In both cases, the detection of duplicate bug reports is useful and needed.

Techniques for automatic detection of duplicate bug reports (e.g., [LM13, RAN07, JW08]) build a model from historical bug reports and detect duplicate bug reports using machine learning techniques. These techniques can be grouped into two main categories based on the features they use to characterize bug reports [BJSWPZ08]. The first category uses the bug report description provided by the submitter. While bug report descriptions can be useful, they remain informal and not quite reliable [RAN07]. The second category consists of the techniques that use stack traces (also called crash traces). Stack traces have shown to be more reliable than bug report descriptions [LM13]. In addition, as noted by Adrian et al. [SBP10], bug reports that have stack traces tend to be fixed sooner.

2.2.2.1. Detection of Duplicate Bug Reports Using Bug Report Description

Techniques that use descriptions to detect duplicate bug reports use words in the description (called terms) as features. They often resort to the use of term frequency and inverse document frequency (TF-IDF) [LM13] to weigh the feature vector constructed by these terms. TF-IDF is a weighting scheme in which, given all terms in a corpus of bug descriptions, it weighs each feature or term by its frequency of appearance in one bug description normalized by its frequency of appearance in the whole corpus [LM13].

Runeson et al. [RAN07] used term frequency applied to the description of bug reports in the bug repository of the Sony Ericsson Company. They used each word in the description as a feature and represented each bug report as a vector. Before calculating the similarity between bug report descriptions, they went through some preprocessing steps. These steps include tokenization, stemming, and stop-words removal. To weigh the features, they used the frequency of each word. They defined an accuracy metric for detecting duplicate bug reports called recall rate. Recall rate is defined as the percentage of duplicate bug reports from the whole duplicate bug reports set in the system for which

their duplicate bug reports were found in the suggested list, provided as the result of the approach. If we assume that the total number of bug reports is T and that the total number of duplicate bug reports for which the duplicates are in the suggested list provided by the approach as D, then the recall rate is calculated as:

$$Recall\ Rate = \frac{D}{T} \quad (1)$$

The authors used various similarity metrics and concluded that different similarity metrics do not change the recall rate significantly. Using the bug report descriptions, they reached a recall rate of up to 42% for detecting duplicate bug reports.

Jalbert et al. [JW08] proposed a linear model to eliminate duplicate bug reports that reach developers. The authors used textual description, categorical information, and clustering information of bug reports as features extracted from the Mozilla project bug repository. They used term frequency to weigh the feature vector. They trained the model to obtain a threshold based on triage and miss effort and used the threshold to eliminate duplicate bug reports. Using the trained model, 10% of duplicate bug reports were eliminated before reaching the developers. They used a leave-one-out approach to show the importance of each feature. They concluded that bug reports titles and descriptions are the most important features to measure the similarity of bug reports.

Duplicate bug reports are not always unwanted since they can provide additional insight into the underlying fault in the software system [BPZK08a]. Thus, detecting duplicate bug reports is not about eliminating them, but to provide additional information about the bugs. Bettenburg et al. [BPZK08a] showed that the developer's performance in fixing bugs could increase if duplicate bug reports are included.

In most bug tracking systems, when a bug is reported, in addition to the textual description of the bug report, the title or a summary of the bug is usually provided. Considering that the title of the bug report usually contains keywords that could have more discriminative power than terms in the description,

Sun et al. [SLWJK10] used a combination of features extracted from the title and the description to train an SVM model to detect duplicate bug reports. The features are 1-Gram and 2-Grams of terms extracted from the summary, the description or both. Then, the inverse document frequency (IDF) is used to weigh the features. The authors showed that the combination of features extracted from both the title and the description of bug reports improves the recall rate by 17%-31% for OpenOffice, 22%-26% for Firefox and 35-43% for Eclipse.

Sun et al. [SLKJ11] showed that having a linear combination of categorical features and textual description features can increase the duplicate bug report detection accuracy of the model compared to models that use only bug report descriptions. The authors also proposed to use BM25 as opposed to TF-IDF to improve the accuracy of the approach. BM25 is a parametrized version of TF-IDF. The parameters in BM25 can be optimized using optimization methods such as gradient descent. BM25 is designed for calculating the similarity of a query to a corpus of documents. Using the optimized duplicate detection method, the authors achieved a recall rate of 45% for OpenOffice, 46% for Mozilla and 53% for Eclipse.

Sureka et al. [SJ10] used a Character-Gram based model instead of a word-based model to detect duplicate bug reports using the description. The advantage of using a character-based N-gram is that it can be used as a free-form text duplicate report detector. This means that the proposed approach is language independent. In addition, character-based N-Gram uncovers key linguistic features of the bug report description. It can handle misspelled words, has the ability to match term variations to a common root and the ability to match hyphenated phrases.

Topic modelling techniques, when applied to a corpus of bug report descriptions, can extract existing topics. Each topic in the context of bug report descriptions can reflect a specific functionality of the system. Nguyen et al. [NNNLS12] proposed DBTM (Duplicate Bug Report Topic Model) that uses topic modelling combined with information retrieval techniques to detect duplicate reports. In DBTM, LDA (Latent Dirichlet Allocation) topic modelling technique and BM25 were parameterized and combined.

The authors optimized the parameter using linear regression. The result shows a 20% improvement over stand-alone informational retrieval techniques.

Alipour et al. [AHS13] used software architecture knowledge to improve duplicate bug reports detection accuracy. They used a word list that is extracted from the software architecture specification as a new feature. Using this contextual information, they improved the accuracy of detecting duplicate bug reports by 12.11%.

2.2.2.2. Detection of Duplicate Bug Reports Using Stack Traces

In addition to bug reports descriptions, stack traces have been used to detect duplicate bug reports. A stack trace ideally consists of a history of function calls starting from the bottom of the stack trace having the first function executed to the top of the stack trace where the last function is executed when the crash happened. This history of function calls gives the triagers and developers clues on which execution path the program went through before it crashed.

Schoter et al. [SBP10] investigated the role of stack traces in helping developers to fix bugs. The authors showed that bugs that are reported with stack traces get fixed sooner. Brodie et al. [BMLSM+05] conducted studies on duplicate bug report detection using stack traces. The authors used stack trace matching algorithms to calculate similarity among stack traces of various bug reports. The authors pruned stack traces before applying stack trace matching algorithms. The same concept of stop-word removal that is implemented in duplicate bug report detection using bug report descriptions can be used to prune stack traces. In stack traces, stop-words are functions that are always called when the software starts (entry-level functions), functions that are called when an error happens (programming language error handling functions), and recursive functions.

When comparing two functions from different stack traces, it is important to consider their position in the stack trace. The functions that are near to the top of a stack trace are more likely to be the cause of failure. Brodie et al. [BMLSM+05] applied the Needleman–Wunsch algorithm (inspired from

bioinformatics) to compare the sequence of functions in stack traces. They showed that comparing each existing stack trace with all the other ones is not possible due to the high computational overhead. They used a B+ tree by hashing top J functions to overcome this problem. They explained that if the top four functions are not the same, the probability of two stack traces being related to the same underlying fault is small.

Modani et al. [MGLMM07] pruned functions from stack traces by removing recursive functions using a function frequency algorithm. The authors used inverted indexing to overcome the problem of high computational overhead when comparing stack traces. They compared three different approaches, including their own, which was similar in principle to the one proposed by Brodie et al. [BMLSM+05], and an approach based on a prefix matching algorithm. The authors showed that the prefix matching algorithm has the best accuracy. They also showed that eliminating uninformative functions increases the accuracy of detecting stack traces that are related to the same underlying fault.

Bugzilla bug tracking system does not have a separate field to store stack traces. Eclipse and Gnome use the Bugzilla bug tracking system. In such systems, users copy the content of stack traces into the bug report description field. Structural information stored in Eclipse Bugzilla is not limited to stack traces but could be patches, source codes or enumerations [BPZK08b]. Bettenburg et al. [BPZK08b] developed a tool called InfoZilla to extract structured information from the bug report descriptions of Eclipse. The authors showed that the accuracy of extracting structural information is 97% in Bugzilla.

Kim et al. [KZN11] suggested creating buckets (clusters) of duplicate bug reports. Each bucket was generalized using a crash graph, which is based on the function calls of all stack traces in a bucket. Having a threshold of 97% similarity, the precision and recall of detecting duplicate bug reports are 68% and 64%, respectively. However, considering that Lerch et al. [LM13] showed that mostly duplicate bug report groups are buckets of size two, the applicability of the proposed approach is expected to be lower for Eclipse bug repository. In practice, when a new bug report is reported, it has only one duplicate bug

report in the dataset, and no generalization can be made on that bug report alone based on crash graph theory. On the other hand, this approach only works when a bucketing system exists in the bug tracking system, which is not the case in many bug tracking systems such as Bugzilla.

Dang et al. [DWZZN12] proposed a method for calculating call stack similarity called the position-dependent model (PDM). The position-dependent model considers two main metrics, namely the distance to the top frame, and the alignment offset to calculate the similarity among stack traces. PDM separates all shared function sequences between two call stacks, and then for each, it calculates the distance to the top and the alignment offset. The evaluation that has been done on five Microsoft products showed that comparing diverse approaches including WER, ReBucketing, Prefix match and Crash Graphs while using buckets created by Microsoft developers as the ground truth, PDM has slightly worse purity than WER but better inverse purity and F-measure on all of the Microsoft products. When PDM was used for bucketing bug reports, the total number of buckets has been reduced by 25%.

Le et al. [LK12], argued against placing only duplicate bug reports in a group. They conducted a study on five Mozilla applications and showed that a good grouping of bug reports should consider reports of bugs that are caused by the same root causes and those that are dependent on these causes. The rationale is that bugs that can be resolved by the same group of developers should be placed in the same group.

Wang et al. [WKZ13] proposed to use crash correlation to group reports of bugs with the related causes in the same crash group. A crash correlation group is defined as a diverse group of crash types that are related to the same faults. Crash types are created by collecting crashes with similar stack traces. Studies on Firefox Bugzilla and Eclipse Bugzilla involving developers showed that using only crash type signatures (the common top frame functions among all the stack traces in a crash type), one can identify the crash correlation groups with an acceptable level of precision and recall. To push the idea to a next step, they showed that not only comparing crash types signatures but also comparing fully qualified file names in

the top frame of crash types or the presence of a frequent close ordered subset of frames between crash types can be used to create crash correlation groups.

Lerch et al. [LM13] applied TF-IDF to the corpus of stack traces considering each frame containing the function name as a feature. Each feature was weighed using its frequency in the current stack trace and normalized by its frequency in the whole corpus (containing all stack traces). The authors used Eclipse bug reports submitted between 2001 and 2008 for their experiments. Their study showed that the majority of duplicate bug reports sets contain only two bug reports. In fact, the number of groups of duplicate bug reports containing only two bug reports is ten times more than the total number of groups of bug reports having more than two bug reports.

Ebrahimi et al. [EH15, EIHH16, ETIHK19] proposed several approaches that leverage the use of generalizable automata and Hidden Markov Models (HMM) and stack traces for the detection of duplicate bug reports. They showed that stack traces can be modeled as stochastic processes, which are used as a prediction model. Their models reach an accuracy of up to 90% using a ranked list of suggested bug reports.

2.2.3. Predicting Bug Severity

When reporting a bug, users usually assign a severity to it. The severity level of a bug should reflect the impact of the bug on the execution of the software system [LDSV11]. Triagers approve or modify the severity level assigned by users. Critical bugs should be dealt with sooner by the developers, whereas low priority bugs could be handled once resources become available. Severity is also used by developers to assign a priority to the bug report. In practice, it is difficult to assign a severity level to a bug without some tool support. There exist several approaches that aim to automatically predict the severity of incoming bug reports based on historical data [GS14, LDGG10, LDSV11].

2.2.3.1. Predicting Severity Using Bug Report Description

Antoniol et al. [AADPKG08] built a dataset using 1800 issues reported to bug tracking systems of Mozilla, Eclipse and JBoss (600 reports from each bug tracking system). In these bug tracking systems, issues can be labeled as corrective maintenance or other kinds of activities such as perfective maintenance, preventive maintenance, restructuring or feature addition. In this study, issues that are related to corrective maintenance are categorized as Bug, while issues related to other activities are categorized as Non-bug. Each of the 1800 issues, extracted from bug tracking systems, is revised and labelled manually. In some cases, bugs were not related to Eclipse, so they are removed from the training and testing set. They considered bug reports descriptions as the best sources of information to train the machine learning techniques for predicting severity [BJSWPZ08]. The authors used words in the descriptions as features. They used the frequency of words for weighing the feature vector, which corresponds to each bug report. In addition to the words in the description, they added the value of the severity field as a feature. After extracting words, they are stemmed. However, no stop-word removal is performed. The rationale behind not removing stop-words is that they may be discriminative and may be used by the classifier to improve classification accuracy. They used various classification methods, such as decision trees, logistic regression, and naïve Bayes to classify issues. Each of the classifiers is trained using the top 20 or 50 features. The accuracy of the approach, when applied to Mozilla, is 67% and 77% with the top 20 and 50 features, respectively. The accuracy of the approach applied to Eclipse issues having 20 features is 81%, and having 50 features is 82%. The accuracy of the approach, when applied to JBoss issues having 20 features, is 80%, and having 50 features is 82%. They showed that for Eclipse, some words (e.g. “Enhancement”) are good indicators of Non-bug issues, while some words (e.g. “failure”) are good indicators of Bug issues. They also showed that their approach outperforms the regular expression based approach, which uses the grep command [AADPKG08].

Menzies et al. [MM08] did a study on an industrial system in NASA. NASA uses a bug tracking system called Project and Issue tracking system (PITS). They examined bug reports raised by testers and sent to

PITS. In NASA severity of bugs are on a five-point scale. One corresponds to the worst, most critical bug and five is the dullest bug. They introduced a tool called SEVERIS, which, using the description of the bug reports and text mining techniques, predicts the severity of an issue. They tokenized terms, removed stop-words and finally stemmed the terms. They used the TF-IDF score of each term to rank them, and then they cut all but top K features. Furthermore, they did another round of feature reduction using information gain. They used rule learner to deduce rules from the weighed features. For the case study, five different systems and consequently, five different datasets are used. The main problem with the datasets was that they did not have any bug with severity one (Critical severity), and the total number of bug reports was 3877. The authors calculated precision, recall and F-measure for each of the severities. Using the top 100 words as features, F-measure was averagely 50% for predicting bugs severities. They showed that in their dataset, using the top 3 features or 100 features does not change the F-measure of the proposed approach significantly. This fact shows that predicting severities using a much smaller number of features that have more discriminative power reveals good results.

Lamkanfi et al. [LDGG10] did a study to show the discriminability power of the terms in bug report descriptions. They build a dataset of open-source software such as Mozilla, Eclipse and Gnome to evaluate the proposed approach. In Bugzilla, severity can be Critical, Major, Normal, Minor, Trivial or Enhancement. To have a coarse-grain categorization, the authors labelled all issues which were Critical or Major as Severe and issues that were Minor, Trivial or Enhancement as Non-Severe. They ignored using issues marked as Normal because it is the default choice that will be assigned to a bug in Bugzilla, and users may choose it arbitrarily. The bug severity prediction in this study is modelled as a document classification problem. They used the summary and description of bug reports for training and evaluation. The authors organized bug reports according to their faulty product and component fields. They used 70% of bug reports as the training set and 30% of bug reports as the testing set. They did a study on the most important features and concluded that words like “crash” or “memory” are good indicators of severe bugs. The authors did a second round of experiments using only descriptions and showed that

using only descriptions decreases the performance in many cases. They did experiment having training sets with different sizes and concluded that a training set having 500 bug reports is enough to have a generalizable result. They also showed that increasing the number of bug reports to more than 500 does not change the evaluation result. In the fourth round of experiments, they showed that when applying the severity prediction approach on the Eclipse bug tracking system, isolating bug reports according to the affected component and product fields leads to a better result.

Lamkan et al. [LDSV11] compared the effect of having diverse mining algorithms applied to bug repositories to predict the severity of the bugs. The authors used the same labeling principal as Lamkanfi et al. [LDGG10]. Eclipse and Gnome are the datasets that are used to evaluate the accuracy of predicted severities. Bug reports are extracted and categorized according to their faulty product and component fields. They compared Naive Bayes, Naive Bayes multinomial, 1-Nearest Neighbour and Support Vector Machine classifiers. Since different classification approaches need different ways to weigh feature vector, in this study when doing experiments using Naïve Bayes, only the presence or absence of each term is used for weighing features. When doing experiments using Naïve Bayes Multinomial, the frequency of each word is used for weighing each feature vector. Using 1-Nearest Neighbour or Support Vector Machine, term frequency and inverse document frequency is used for weighing feature vectors. They calculated precision and recall, the area under the curve (AUC) for each dataset to compare the classifiers. They showed that using Naïve Byes Multinomial, the area under the curve is averagely 80% which is higher than other approaches. Next, they showed that using the Naïve Bayes classifier, a stable accuracy value is achieved having 250 bug reports of each severity for training. The result shows that increasing the training set by adding more than 250 bug reports does not change the accuracy. Having a list of words that have good discriminative power, they concluded that each component has its list of words. Thus, terms that have good discriminative power are component-specific. This result encourages applying severity prediction approaches on each component independently since it reveals better results.

Yang et al. [YHKC12] compared the effectiveness of feature selection methods on a coarse grain severity prediction technique. They used the Naïve Bayes classifier to study the effectiveness of each feature selection method. They used information gain, Chi-square and correlation coefficient as feature selection techniques. They used Eclipse and Firefox datasets and true positive rate (TPR), false positive rate (FPR) and area under curve (AUC) metrics to evaluate their studies. They showed high information gain is a good indicator of severe bugs and low information gain is a good indicator of non-severe bugs. They concluded that the best feature selection technique for Eclipse and Mozilla is the correlation coefficient. While all of the studies presented so far focus on a coarse-grain severity prediction, Tian et al. [TLS12] proposed an approach for finer grain prediction of bug severities. A fine-grain approach targets each severity type (Critical, Major, Normal, Minor, Trivial, and Enhancement) to predict the severity of a bug by classifying it into one of these types. The authors used OpenOffice, Mozilla and Eclipse bug repositories to evaluate their approach. In the first place, they removed bug reports having Normal severities, because they are the default value when reporting a bug. They used the K-nearest neighbour algorithm to predict severity. They used $BM25_{ext}$ as the similarity metric. They used the similarity of bug reports descriptions using 1-gram words as the first feature, similarity of the descriptions of bug reports using 2-gram as the second feature, and faulty product and component fields of bug reports as the third and fourth features respectively. With the arrival of each new bug report, it is compared to all the existing bug reports and the severity of the bug that is the closest to the incoming bug is used as the predicted severity of the incoming bug.

Yang et al. [YZL14] studied the effectiveness of topic modeling on fine-grain severity prediction. Instead of using categorical features in similarity calculation, they only considered bug reports if they had the same product, component and priority. In the first step, they used latent Dirichlet allocation (LDA) to extract topics from the corpus of documents. They represented each topic as a bag of words. Instead of the vector space model, they used smoothed unigram vectors and they used KL divergence instead of cosine similarity to measure the similarity of smoothed vectors. They showed the effectiveness of their

approach by applying it on Mozilla, Eclipse and NetBeans bug repositories. They used an online approach in which with the incoming of each bug report its probability vector is extracted. K-nearest neighbour is then applied to the bug reports with the same topics and according to the returned list of similar bug reports, the label is chosen.

Bhattachrya et al. [BINF12] explored alternate avenues for bug severity prediction using a graph-based analysis of the software system. They built a graph based on different aspects of software systems including source code graphs based on function calls (e.g., a static call graph) or modules (module collaboration graph) and, at a more abstract level, they built a graph based on developer's collaboration. They used Firefox, Eclipse and MySql to show the effectiveness of their approach. They used various graph-based metrics including average degree, clustering coefficient, node rank, graph diameter and assortativity to characterize software structure and evolution. They showed that the node rank metric in the function call graph is a good indicator of bug severity. They also showed that the Modularity Ratio is a good indicator for modules that need less maintenance effort.

Zhang et al. [ZYLC15] explored the effectiveness of concept profiles in predicting the severity of bugs. They extracted concept terms and calculated their threshold for each fine-grain severity label in the training set. A concept profile corresponding to each severity label using concept terms is built next. Instead of the vector space model, they used the probability vector to represent each bug report. Also, instead of cosine similarity, they used KL divergence for calculating the similarity between each bug report in the testing set and each concept profile which corresponds to each severity label in the training set. They showed the effectiveness of their approach by applying it on Eclipse and Mozilla bug repositories. They used an offline approach in which they used 90% of the data as the training set and 10% of the data as the testing set.

2.2.4. Bug Report Faulty Product and Component Field Prediction

Faulty components or products are important fields of bug reports that are used by triagers and developers to localize the faults. When submitting a bug report, a user has the option to choose a product of the software system from a predefined set provided by the bug tracking system. Since each product, in the system, has its own set of components, after choosing the product, a set of predefined components according to the selected product will be made available to the user to choose from.

2.2.4.1. Bug Report Faulty Product and Component Field Prediction Using Bug Description

An early work on bug categorization was conducted by Lucca et al. [LPG02]. The authors used five different text mining models to classify incoming bug reports (referred to as tickets in their paper). They manually labelled each ticket into eight predefined categories, each corresponding to a maintenance team. They used the description of tickets and applied a probabilistic model, a vector space model, support vector machine, regression tree, and K-nearest neighbour. They showed that probabilistic and the K-nearest neighbour models provide the best accuracy.

Betteneburg et al. [BJSWPZ08] explained that since users do not usually have technical knowledge about the system, it is very difficult for them to properly report the faulty product and component fields. They also showed that incomplete (or incorrect) information of a bug report is one of the major obstacles for developers for providing fixes.

Guo et al. [GZNM11] showed that there are five main reasons which cause bug report field reassignment: Finding the root cause, determining ownership, poor bug report quality, proper fix determination (identifying how to fix the root cause), and workload balance. They introduced the bug pong concept as sending the bug reports between development teams similar to ping pong ball. The authors showed that it happens when a faulty component field is not correctly identified in the bug report. The authors also showed that the incorrect selection of the faulty component field increases the bug reports processing time.

Breu et al. [BPSZ09] showed that the questions asked when submitting bug reports can be grouped into eight categories: Missing information, clarification, triaging, debugging, correction, status inquiry, resolution, and administration. They showed that triaging questions are mostly due to the fact that users enter the wrong faulty product and component when reporting a bug.

Somasundaram et al. [SM12] showed that the component field of bug reports help triagers to route bug reports to the right development team. They also showed that incorrect component categorization often delays the resolution of the bug reports. They used the description of bug reports and applied Support Vector Machine (SVM), Latent Dirichlet Allocation (LDA) with SVM and LDA with Kullback divergence (KL). They showed that LDA-KL produces more stable results than SVM or LDA-SVM.

Bhattacharya et al. [BN10] proposed a novel approach based on a multi-feature tossing graph to improve the bug triaging process. They trained their classifiers on the description, title, faulty product, and component fields of bug reports. They showed that adding a multi-featured tossing graph improves the bug report routing accuracy.

Shihab et al. [SIKIO+10] showed that the processing time of re-opened bug reports is two times more than the processing time of regular bug reports. They showed that description, fixing time, and the faulty component field of bug reports are the most important factors in determining whether a bug report will be re-opened. They showed that the bug reporter name is not an important factor in predicting re-opened bug reports. They structured their study in four dimensions: Work habit (weekday of closing bug reports), bug reports faulty component fields, bug fix (time spent to fix the bug), and people (experience of the bug fixer). They extracted features from these dimensions to build a decision tree for predicting whether a bug report will be re-opened.

Giger et al. [GPG10] showed that, in Firefox bug reports, the faulty component field is the most important factor in determining how fast a bug will be fixed. They also showed that, in Gnome bug reports, the faulty component field is one of the most important fields in determining the fixing time of the bug.

Sureka [S12] showed that the most important feature to localize the fault of a bug is the component. He also showed that the component field is usually assigned incorrectly by most users. He showed that the highest frequency of reassignment after the assignee field is the component field. The author used a statistical, and probabilistic model applied to the title and description of bug reports to predict the faulty component field. The study achieved 42% accuracy when predicting the faulty component field of bug reports. The author also showed that TF-IDF applied to the description of bug reports performs the same as the Dynamic Language Model (DLM). His approach predicted the reassignment of the component field of bug reports with 42% accuracy.

Lamkanfi et al. [LD13] showed that the faulty component field in Eclipse and Mozilla bug reports tends to be regularly reassigned. They extracted all initial values of a bug report information including component, reporter, operating system, version, severity, and bug report summary to decide whether the component field of the bug report will be reassigned. They trained a Naïve Bayes classifier. Their classifier predicted reassigned bug reports with an F-measure of over 44% and not reassigned bug reports with an F-measure of over 83%.

Xia et al. [XLWSZ14] showed that 80% of bug reports have their fields reassigned. They also showed that the bug reports that are reassigned take a longer time to be fixed. They showed that the product and component fields usually get reassigned together. The authors also showed that some product and component fields reassignments occur more than three years after the initial submission.

Xia et al. [XLSW16] used a multi-label learning algorithm (ML.KNN) to predict the reassignment of bug report fields. To overcome the unbalanced dataset problem, they used the IM-ML.KNN classification approach. IM-ML.KNN is a composite classifier that is used as a combination of three classifiers which are built based on metadata of bug reports, textual information of bug reports, and a mixture of both (metadata and textual information). Their approach achieved an accuracy (F-measure) ranging from 56% to 62%.

Wang et al. [WZLLW12] used bug report descriptions and summaries to predict the component field of bug reports. They created feature vectors using words in the descriptions and summaries. They weighed the feature vectors using TF-IDF. They compared the performance of support vector machine and Naïve Bayes machine learning techniques. They applied their approach to Eclipse bug reports and showed that the support vector machine model outperforms the Naïve Bayes model in predicting components.

Florea et al. [FAA16] introduced a parallel recommender system to help assign bug reports to the right developers. They used textual description and categorical information as features and convolutional and recurrent neural networks (CNN and RNN) to predict the developers that should fix the bug. The authors applied their approach to Netbeans, Eclipse and Mozilla datasets. They showed that their parallel deep learning-based implementation using CNN has comparable results to multiple SVMs running in parallel.

Zhang et al. [ZWW15] introduced an approach that assigns bug reports to developers using a two-phase process. In the first phase, the K-nearest neighbour is applied to the history of bug reports. The second phase ranks developers based on their experience with similar bugs. The authors built a heterogeneous network based on five entities (developers, bugs, comments, components and products) and ten types of relations among these entities (e.g., reported by, commented by, written by, etc.). Bug report descriptions are transformed into vectors weighed using TF-IDF. For each bug report, the K-nearest bug reports with the same descriptions are retrieved, and a second refinement ranking is done to rank developers based on the network. They tested their approach on Apache Ant, Apache Tomcat, Mozilla, and Eclipse and showed that their approach improves recall by 7.5% to 32.25% on those datasets.

Chapter 3

Data Preparation

In this chapter, we discuss the datasets we used to evaluate the contributions of the thesis.

3.1. Eclipse Dataset

In this thesis, we used Eclipse and Gnome datasets to evaluate the proposed approaches. Eclipse is an integrated development environment which is written in java. Eclipse bug repository has evolved during the past 16 years. It has been the subject of mining challenge in the Mining Software Repositories 2008 conference. It is a platform in which various plugins can be added to it for different purposes. One of the main reasons for choosing Eclipse is that it is an open-source system which incorporates a large number of developers. Every day large number of bug reports are being sent to triagers in the Eclipse bug tracking system. Having more than 500,000 reports starting from 2001 to 2015 makes Eclipse a comprehensive dataset that can be used for evaluating our proposed approaches.

Since in the current version of Bugzilla, users can not send the structural information including stack traces of bug reports separately, they have to copy that information into the description of the bug reports. In MSR 2008, a tool called Infozila was proposed by Bettenburg et al. [BPZK08b]. This tool can extract four types of structural information including stack traces, source codes, patches, and enumerations from Eclipse bug report descriptions. Lerch et al. [LM13] provided some enhancement to the regular expression which was provided by Bettenburg et al. [BPZK08b] and proposed the following regular expression to extract stack traces more accurately.

```
[EXCEPTION]    ([:][MESSAGE])?    ([at][METHOD][(]
[SOURCE] [)] )+ ( [Caused by:] [TEMPLATE] )?
```

Figure 5 Regular expression used for extracting stack traces from bug report descriptions in the Eclipse bug repository

Our Eclipse dataset consists of Eclipse bug reports from October 2001 to February 2015. In this period, 454,486 reports were submitted to the Eclipse bug tracking system. Out of these reports, 66,859 were labelled as Enhancement. 42,892 bug reports from the 454,486 bug reports (11%) in Eclipse had at least one stack trace in their description.

Table 1 Eclipse dataset charchteristics

Data	Eclipse
Total number of reports	454,486
Total number of bug reports	387,627
Total number of enhancement reports	66,859
Total number of bug reports with stack traces	42,892

3.2. Gnome Dataset

Gnome is a desktop environment that is designed for Linux and BSD based operating systems. The Gnome bug repository contains more than 17 years of bug reports. It has been the subject of challenge in the Mining Software Repositories 2009. Gnomes' main purpose is to provide a powerful and easy to use desktop environment to users who are using Linux or BSD based operating systems. Gnome used to have the same bug tracking system as Eclipse (From March 2019 Gnome started using GitLab instead of Bugzilla). Similar to Eclipse, it did not have any separate mechanism for storing structural information. The main difference between Gnome and Eclipse is that Gnome is written using multiple programming languages. Most of the Gnome is written in C and the rest is written in other programming languages such as C++, Perl, Python, and Lisp [FFHL05]. Since Infozilla is designed to capture stack traces that are written Java, it cannot be used to extract stack traces from the Gnome bug tracking system. We developed a tool that extracts all four types of structural information (stack traces, source codes, patches, and enumerations) from Gnome bug reports. We used the regular expression of Figure 6 to extract stack traces from the description of the Gnome bug reports.

```
([#NUMBER] [HEX ADDRESS] [IN] [FUNCTION NAME] [(]
[PARAMETERS] [)] ([FROM] | [AT]) ([LIBRARYNAME] |
[FILENAME]))*
```

Figure 6 Regular expression used for extracting stack traces from bug report description in the Gnome bug repository

We verified our tool by manually examining structural information extracted from 1000 Gnome bug reports. The precision and recall of our method for extracting stack traces is 99% and 90%, respectively. We have a precision and recall of 97% and 100% for detecting source codes. For patches, the precision and recall are 100% and 100% respectively. Our tool could extract Enumerations with a precision and recall of 80% and 95% respectively.

Our Gnome dataset consists of Gnome bug reports from October 1999 to August 2015. In this period, 697,800 reports were submitted to the Gnome bug tracking system. Out of these reports, 57,446 were labelled as Enhancement. 201,302 bug reports from the 640,354 bug reports (32%) in the Gnome bug repository had at least one stack trace in their description.

Table 2 Gnome dataset charchteristics

Data	Gnome
Total number of reports	697,800
Total number of bug reports	640,354
Total number of enhancement reports	57,446
Total number of bug reports with stack traces	201,302

Chapter 4

An Empirical Study on the Effectiveness of Presence of Stack Traces in Bug Triaging Process

Some studies systematically investigate the usefulness of stack traces in debugging software bugs and in fixing time of bugs [SBP10, K12, YCKY14, MHNSL15]. Schorter et al. [SBP10] studied the association between stack traces and source code debugging. They showed that bug reports with stack traces are fixed sooner. Krikke [K12] and Yang et al. [YCKY14] studied the association between the presence of stack traces and the value of bug report fields such as the severity field.

In this chapter, we shed light on newer aspects of the usefulness of stack traces in the bug triaging process. More precisely, we extend Schroter et al. [SBP10], Krikke [K12] and Yang et al. [YCKY14] studies by examining the relationship between the presence of stack traces and the time to detect duplicate bug reports. We also extend previous studies by studying the association between stack traces and the severity of bug report in the Gnome bug repository as well as the reassignment of product and component fields of bug reports.

4.1. Dataset Setup

We used the Eclipse bug reports from 2010-1-1 to 2014-1-1 and Gnome bug reports from 2010-1-1 to 2015-1-1 for this study. Although we chose a window of bug reports from the datasets presented in Chapter 3, we believe that the conclusions of this study reflect the overall characteristics of the dataset irrespective of the selected time window because of the use of statistical tests.

In Bugzilla, each bug report page is linked to a bug activity page, which shows changes to the bug report fields. More precisely, it shows what was done (e.g., bug report product field was reassigned), who did it, and when did it happen. We use the bug activity page to extract resolution time (fixing time) of duplicate reports, product and component field reassignments, and fixing time of bugs.

Bugs in Gnome and Eclipse have seven severity levels. We remove bugs with normal severity since they represent the grey area and are usually selected arbitrarily by the users [YCKY14]. Then, we categorize the bug reports into two groups: Severe bugs include bugs with Blocker, Critical and Major severity and Non-severe bugs include bugs with Minor and Trivial severities [YCKY14].

4.2. Statistical Analysis

We present our data collection and analysis process in Figure 7. For each bug report, we extract the following information.

- The presence of stack trace in the bug report
- The bug report opening time and fix closing time
- The bug report severity value
- If the product field of the bug report is reassigned
- If the component field of the bug report is reassigned

We use this information to calculate the fixing time of duplicate reports, coarse grain severity level, and the reassignment of the bug report faulty product and component fields.

We use two-tailed Mann-Whitney test to calculate if the difference between two sets of results (e.g., the duplicate bug report fixing time of bugs with stack traces and those without stack traces) is statistically significant. We use two-tailed Chi-square test to show if the association between two variables is statistically significant (e.g., presence of a stack trace in a bug report and the severity of the bug).

4.3. Experiments

We explain our experiments by listing the research questions and presenting the variables and the analysis method used to answer these research questions. For each research question, we define a null hypothesis corresponding to that research question.

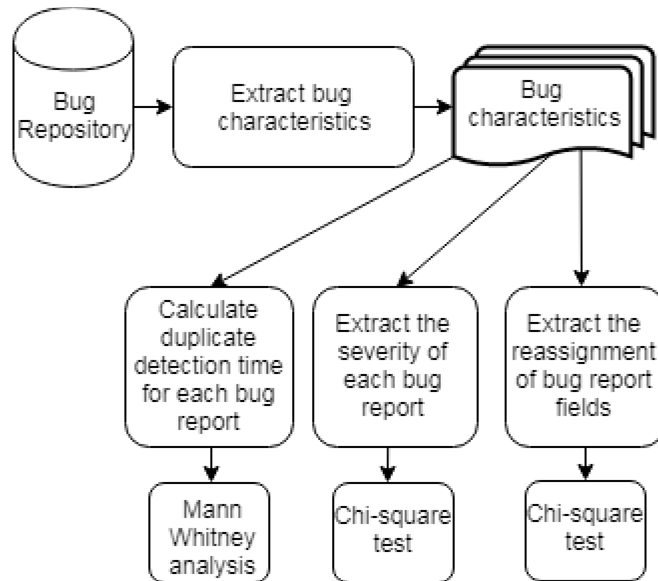

Figure 7 Data collection and analysis

RQ1: Does the presence of stack traces in bug reports reduce the duplicate detection time of bug reports?

Lerch et al. [LM13] showed that stack traces can be used to detect duplicate bug reports with higher accuracy compared to bug report descriptions. Triagers and developers mark a new bug report as fixed duplicate if they find the bug report to be a duplicate of previously seen bug reports. If triagers and developers leverage stack traces for detecting duplicate bug reports, we expect bug reports with stack traces to have lower duplicate detection time compared to bug reports without stack traces.

$\boldsymbol{H_{01}}$: There is no statistically significant difference between the duplicate detection time of bug reports with stack traces and those without stack traces.

Variables: Our variables to answer RQ1 is the fixing time of duplicate bug reports and a boolean that indicates the presence of a stack trace in the bug report. We use the period between the opening time and the time that the bug report is marked as duplicate fixed to calculate the duplicate bug report fixing time. We perform the test of independence between the observed variables (duplicate detection fixing time of bug reports with and without stack traces).

Analysis method: We answer RQ1 in two steps. In the first step, we use the Mann-Whitney test to assess if there is a statistically significant difference between the duplicate detection fixing time of bug reports with stack traces and those without stack traces.

RQ2: Is the presence of stack traces in bug reports associated with the severity level of bugs?

Bug severity is an indicator of how much the bug affects the normal execution of the software system [LDGG10]. In this question, we want to know if there is an association between the presence of stack traces and the type of severity of bugs.

H_{02}: Bug severity type is statistically independent of the presence of stack trace in the bug report.

Variables: Our variables are the bug severity type and a boolean variable indicating whether the bug report has a stack trace or not. We categorize bugs based on two severity types (severe or non-severe). We also categorize bug reports based on the presence of stack traces. Then, we build the contingency table using the severity types and the presence of stack traces. Our contingency table contains the number of severe bug reports with and without stack traces and the number of non-severe bug reports with or without stack traces.

Analysis method: We answer RQ2 using the Chi-square test of independence. More precisely, we use Chi-square test to test the independence of the two variables: bug severity type and the presence of a stack trace. We also analyze the type of severity of bug reports with stack traces and type of the severity of bug reports without stack traces. We show the percentage of severe bugs with and without stack traces and the percentage of non-severe bugs with and without stack traces.

RQ3: Does the presence of stack traces help to identify the right product and component fields of bug reports? Does this improve the fixing time of bugs?

Studies have shown that bug reports which their fields are reassigned take a longer time to be fixed [XLSW16]. In particular, product and component fields of bug reports can help triagers direct bug reports to the right development team. Bug fixing time can also be improved by correctly choosing product and component fields. In this question, we want to know if there is an association between the presence of stack traces and the reassignment of the product and component fields of bug reports. We also study the difference between the fixing time of bug reports, which have their product and component field reassigned, with stack traces and without stack traces. We state the following hypotheses:

$\boldsymbol{H_{03A}}$: Reassignment of the product field of bug reports is statistically independent of the presence of stack traces in the bug reports.

$\boldsymbol{H_{03b}}$: Reassignment of the component field of bug reports is statistically independent of the presence of stack traces in the bug reports.

$\boldsymbol{H_{03c}}$: There is no statistically significant difference between the fixing time of bug reports, which have their product field reassigned, with stack traces and without stack traces.

$\boldsymbol{H_{03d}}$: There is no statistically significant difference between the fixing time of bug reports, which have their component field reassigned, with stack traces and without stack traces.

Variables: Our variables consist of a boolean variable that indicates if the product (or component) of a bug report has been reassigned (its value changed after the report has been submitted), the fixing time of these bug reports, and another boolean variable that indicates if the bug report has a stack trace or not. For each bug report, we parse the history of changes to the bug report fields. We label a bug report field as reassigned if the value of that field is changed in the bug activity history. We consider the time between the opening time of a bug report and marking the bug report fixed as the fixing time of a bug report. We use the reassignment of the product field of bug reports and the presence of stack traces in those bug reports to build a contingency table. We also use the reassignment of the component field of bug reports and the presence of stack traces to create another contingency table. We build two lists

containing fixing time of bug reports, which have their product field reassigned, with and without stack traces. We also build two other lists for the component field with the same approach.

Analysis method: We answer RQ3 in four steps:

1. We use the two-tailed Chi-square test to test the independence of the reassignment of the product field and the presence of the stack traces.
2. We use the two-tailed Chi-square test to test the independence of the reassignment of the component field and the presence of the stack traces.
3. We use two-tailed Mann-Whitney test to test if the difference between the fixing time of bug reports with and without traces and for which their product field is reassigned is statistically significant. If two sets are statistically different, we use the median to compare them together.
4. We use two-tailed Mann-Whitney test to test if the difference between the fixing time of bug reports with and without traces and for which their component field is reassigned is statistically significant. If two sets are statistically different, we use the median to compare them together.

4.4. Study Result

RQ1: Based on two-tailed Mann-Whitney test, the time to detect a duplicate bug report and mark it as duplicate fixed is significantly different for bug reports with stack traces compared to those without stack traces with p-value < 0.05 for Gnome and Eclipse datasets. Since p-value < 0.05, we can reject the null hypothesis H_{01} that there is no statistically significant difference between the duplicate detection time of bug reports with stack traces and the ones without stack traces.

To further examine which group has lower fixing time, we compared the median of the number of minutes it took to fix duplicate bug reports with stack traces to the median time it took to fix bug reports without stack traces. For Eclipse, the median duplicate fix time of bug reports with stack traces is 221.8333 minutes (0.15 day), whereas the median duplicate fix time of bug reports without stack traces is 253.4333 minutes (0.17 day).

For the Gnome dataset, the median of duplicate bug reports fix time is 17748.6 minutes, which is 12.33 days for bug reports with stack traces and 18844.475 minutes (13.086 days) for bug reports without stack traces.

These results confirm that the duplicate detection fix time for bug reports with stack traces is statistically lower compared to duplicate detection fix time of bug reports without stack traces. We can conclude that the presence of stack traces in bug reports has an impact on the time it takes to detect duplicate bug reports.

RQ2: We applied the two-tailed Chi-square test to test the independence of two variables: The presence of stack traces and the severity of the bugs. We applied the test on the contingency table containing the number of severe bug reports with stack traces, the number of severe bug reports without stack traces, the number of non-severe bug reports with stack traces, and the number of non-severe bug reports without stack traces.

For Eclipse, Chi-square test result is 14.021 with 1 degree of freedom. The two-tailed p-value = 0.0002. Since p-value < 0.05, we can reject the null hypotheses H_{02} that the two variables are statistically independent. The chi-square test result shows that there is a relationship between the severity of bugs and the presence of stack traces in the bug reports. This is consistent with the result presented by Yang et al. [YCKY14].

For Gnome, the Chi-square test result is 101.470 with 1 degree of freedom. The two-tailed p-value = 0.0001. Since p-value < 0.05, we can reject the null hypotheses H_{02} that the two variables are statistically independent. Similar to Eclipse, we conclude that there is a relationship between the severity of bug reports and the presence of stack traces in the Gnome dataset. Figure 8 and Figure 9 show the percentage of severe and non-severe bug reports based on the presence of stack traces for Eclipse and Gnome datasets respectively.

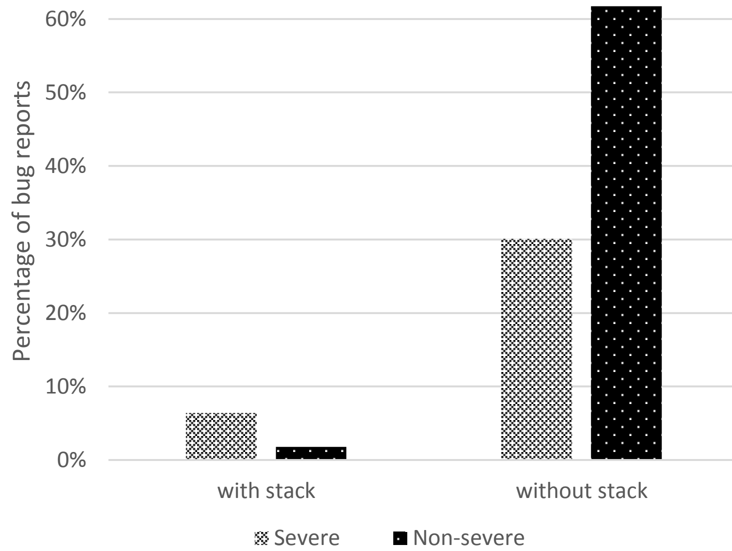

Figure 8 Percentage of severe and non-severe bug reports based on the presence of stack traces in Eclipse dataset.

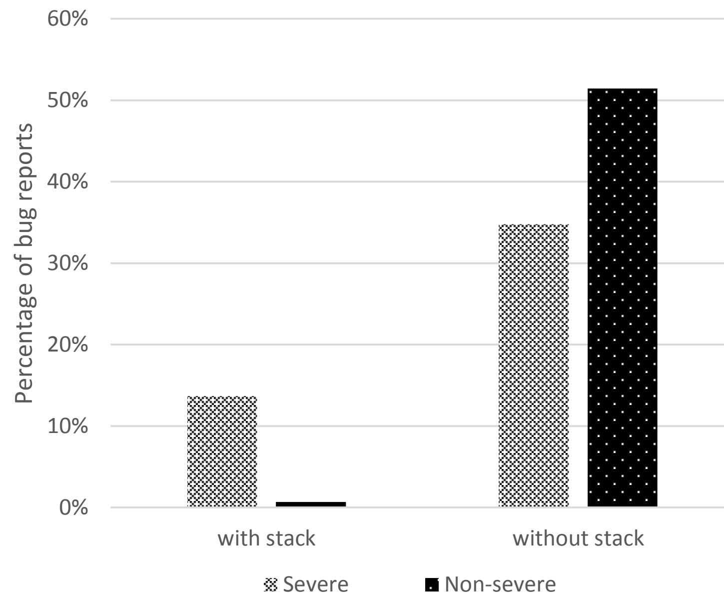

Figure 9 Percentage of severe and non-severe bug reports based on the presence of stack trace in Gnome dataset

To examine the type of severity of bug reports with stack traces and the bug reports without stack traces, we calculate the conditional probability that a bug report has high severity if it has stack trace and the conditional probability that a bug report has high severity if it does not have stack trace.

For Eclipse, the conditional probability of a bug report to have a high severity level if it has a stack trace is 78.4%, whereas the probability of a bug report to have high severity, if it does not have stack trace is 32.7%. For Gnome, the conditional probabilities are 98% and 40%. The result shows that bug reports with stack traces tend to have higher severity levels.

RQ3: We apply two-tailed Chi-square test on two contingency tables. The first contingency table contains the number of bug reports with reassigned product field with and without stack traces. The second contingency table contains the number of bug reports with reassigned component field with and without stack traces.

For Eclipse, the Chi-square test result on the contingency table based on the reassignment of the product field is 31.80 with 1 degree of freedom. The two-tailed p-value = 0.0001, which is less than 0.05. We can reject the null hypotheses H_{03A} that the two variables are statistically independent. For Gnome, the Chi-square test result is 93.43, with 1 degree of freedom. The two-tailed p-value = 0.0001, which is less than 0.05, so we can reject the null hypotheses H_{03A} that the two variables are statistically independent. We conclude that there is a relationship between the reassignment of the product (and component) field of a bug report and whether the bug report contains a stack trace or not.

Figure 10 and Figure 11 show the percentage of bug reports with and without reassigned product field in relation to the presence of stack traces for Eclipse and Gnome datasets, respectively. These figures show clearly that the presence of stack traces has an impact on the reassignment of the product field.

As for the component field, for the Eclipse dataset, the Chi-square test result is 31.806 with 1 degree of freedom. The two-tailed p-value = 0.0001, which is less than 0.05. We can reject the null hypothesis H_{03b}. For Gnome, the Chi-square test result is 93.43 with 1 degree of freedom. The two-tailed Chi-square p-value = 0.0001, which is less than 0.05. Similar to Eclipse, we can reject the null hypothesis H_{03b}. These results show that there is a relationship between the reassignment of the bug reports component field and the presence of stack traces.

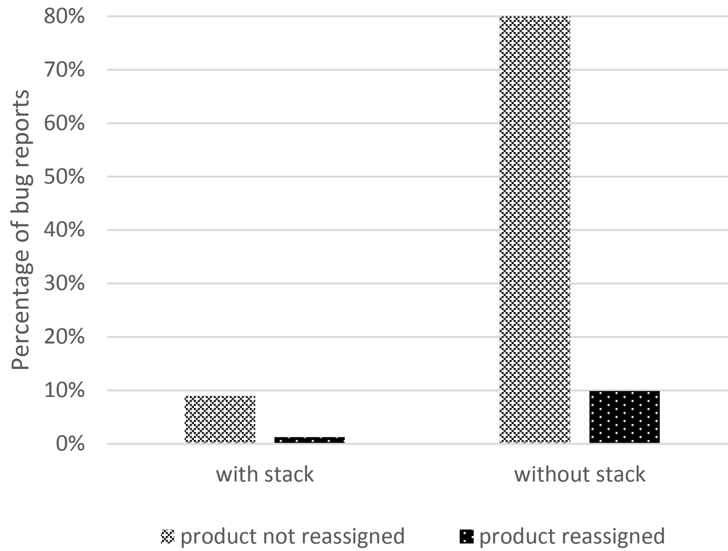

Figure 10 Percentage of bug reports based on the presence of stack traces in the Eclipse dataset

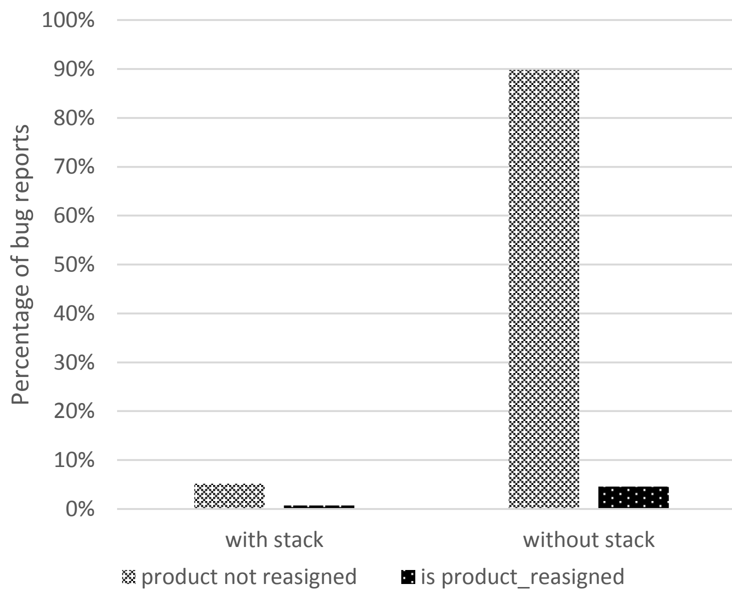

Figure 11 Percentage of bug reports based on the presence of stack traces in the Gnome dataset

Figure 12 and 13 shows the percentage of bug reports with and without reassigned components in relation to the presence of stack traces in Eclipse and Gnome, respectively. The results show that the presence of stack traces has an impact on whether a bug report component field is reassigned or not.

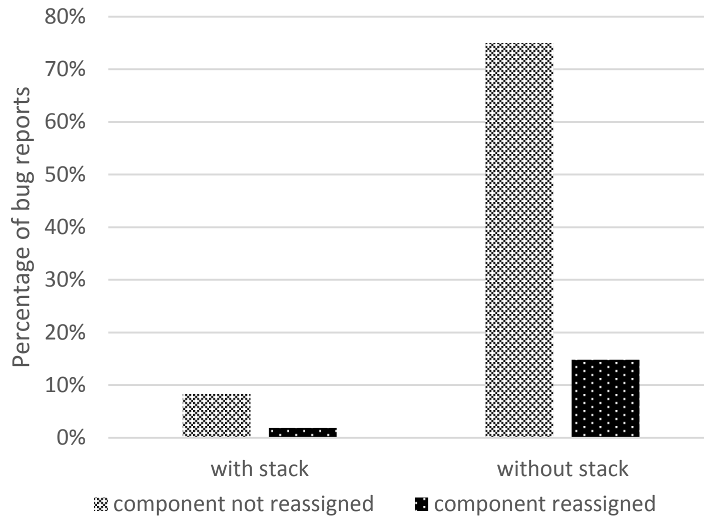

Figure 12 Percentage of bug reports based on the presence of stack trace in the Eclipse dataset.

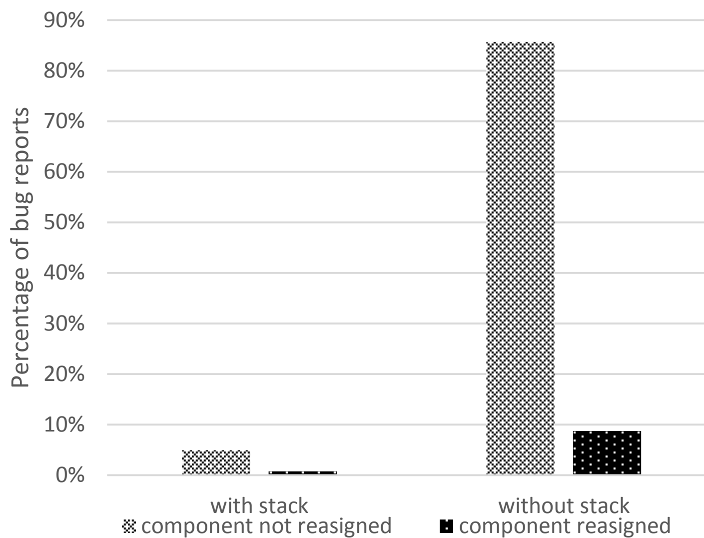

Figure 13 Percentage of bug reports based on the presence of stack trace in the Gnome dataset.

To calculate if the probability of reassigning the product field is higher or lower in the presence or not of stack traces, we calculate the conditional probability of bug reports with their product field reassigned depending on whether they have a stack trace or not. For Eclipse, the probability that the product field of a bug report gets reassigned if the bug report has a stack trace is 12%. It is 10% in the absence of a stack trace. For Gnome, the results are 7% and 4%. We conclude that, for both Eclipse and Gnome, the probability that the product field gets reassigned is higher for bug reports with stack traces compared to bug reports without stack traces.

For Eclipse, the probability that the component field of a bug report gets reassigned if the report has stack trace is 18.4% compared to 16.5% in the absence of a stack trace. For Gnome, the probabilities are 12.5% and 9.2%. The result shows that the probability of reassigning the component field is higher for bug reports with stack traces compared to those without stack traces.

Moreover, we study the time to fix bug reports with reassigned product (and component) field depending on whether the report has a stack trace or not. We use the two-tailed Mann-Whitney test to assess the statistical significance of the difference between the fixing time of the two groups of bug reports.

The results show that, for the Eclipse dataset, the time to fix bug reports with the reassigned product (or component) fields and stack traces is significantly different from the fixing time of bug reports without stack traces with a p-value < 0.05. We can reject the null hypothesis H_{03c} and H_{03d} and conclude that there exists a significant statistical difference between the fixing time of bug reports which their product (or component) field is reassigned with or without stack traces. For Gnome there exist a statistically significant difference between the fixing time of bug reports which their component field is reassigned with or without stack traces, however, there is no significant difference between the fixing time of bug reports which their product field is reassigned with or without stack traces.

The median of the time to fix Eclipse bug reports with reassigned product field is 14444.85 minutes (10 days), whereas the median of time to fix of bug reports with their product field reassigned and do not have stack traces is 10324.8 minutes (7.17 days). These medians are 17405.9 minutes (12 days) and 28853.1 minutes (20 days) for the component field.

For Gnome, the median of the fixing time of bug reports with reassigned component fields with stack traces is 26031.53minutes (18 days), and the median of the fixing time of bug reports with reassigned component field and without stack traces is 29961.7 minutes (20.8 days). Since no significant difference between the fixing time of bug reports which their product field is reassigned with or without stack traces

for the Gnome dataset, we did not compare the median of time to fix of bug reports which their product field reassigned with or without stack traces.

4.5. Discussion

We studied the effect of the presence of stack traces on various aspects of bug reports. We showed that bug reports with stack traces take less time to be detected as duplicates, compared to bug reports without stack traces. We also showed that consistent with previous findings on the Eclipse dataset [YCKY14], there is a relationship between the presence of stack traces and the severity type of bug reports in the Gnome dataset. In addition, we showed that there is a relationship between the presence of stack traces and the reassignment of product and component fields in bug reports. We also showed that bug reports with reassigned product or component fields and stack traces take less time to be fixed, compared to bug reports (with reassigned product and component fields) without stack traces. The results suggest that stack traces may be one factor for determining the accurate product and component fields.

However, we want to note that there are many other factors that can affect the bug fixing time, including the experience of developers, the accuracy of bug report descriptions, etc. Therefore, in this thesis, we can only state that the presence of stack traces could be one factor that can characterize bug reports in a way that may speed up the fixing time. We also need to examine the extent to which stack traces can be helpful in comparison with other bug report elements such as the descriptions, the developer experience, etc. We defer this to future work.

Chapter 5

Detecting Duplicate Bug Reports

There exist techniques for automatic detection of duplicate bug reports (e.g., [DMJ11, RAN07, JW08]). They build a model from historical bug reports, using machine learning techniques. The model is then used to detect duplicate bug reports. These techniques can be grouped into two main categories based on the features they use to characterize bug reports. The first category uses the bug report description provided by the submitter. They assume that bug reports with similar descriptions are potential duplicates. While bug report descriptions can be useful, they remain informal and not quite reliable [JW08]. The second category encompasses the techniques that use stack traces and assume bug reports with similar stack traces are duplicate. Stack traces have shown to be more reliable than bug report descriptions [LM13] to detect duplicate bug reports. In addition, as noted by Schroter et al. in [SBP10], bug reports that have stack traces tend to be fixed sooner.

A recent technique for detecting duplicate bug reports using stack traces was proposed by Lerch et al. [LM13]. The authors developed a technique for comparing stack traces of different bug reports. They considered bug reports with similar stack traces as potential duplicates. They used TF/IDF (Term Frequency-Inverse Document Frequency) as a weighing mechanism. Their approach, however, suffers from scalability problems due to the large number of distinct functions that exist in large bug repositories.

In this chapter, we propose a more efficient approach for detecting duplicate bug reports using stack traces, called DURFEX (Detection of Duplication Reports Using Feature Extraction). DURFEX uses a multi-step feature extraction algorithm that leverages the concept of trace abstraction. More precisely, we substitute functions in historical stack traces by their package names to reduce the number of features used for building the training model. We then use the varying length N-gram of packages to build feature

vectors from stack traces. To detect whether an incoming bug report is a duplicate of an existing one, first, we measure the distance between the vector of an incoming bug report and the bug reports in the bug tracking system, then we use the linear combination of the stack traces similarity and non-textual (categorical) features to retrieve a list of potential duplicate bug reports.

This chapter is organized as follows: We discuss our feature extraction and weighing techniques in Section 5.1. The overall approach and detailed implementation of the approach is presented in Section 5.2. The evaluation techniques are presented in Section 5.33. We discuss the results of our approach in Section 5.4. Threats to validity are presented in Section 5.5. The conclusion is presented in Section 5.6.

5.1. Preliminaries

This section starts by providing a formal description of term vector weighing strategy, term frequency and inverse document frequency (considered in related work [LM13]). The proposed approach for extracting feature vectors from traces of package names based on variable length N-grams is described next.

Let $T = p_{1,}p_{2,} \ldots\ldots, p_L$ be a stack trace, *T* of length *L*, where the function calls are substituted by their defining packages. The stack trace *T* is generated by a bug in a system with an alphabet Σ of size *m* = $|\Sigma|$ (unique) package names. The collection of *K* traces that are generated by the process (or system) of interest and then provided for designing the duplicate detection system is denoted by $\Gamma = T_{1,}T_{2,} \ldots\ldots, T_K$.

The term vector maps each trace $T \in \Gamma$ into a vector of size m packages, $T \rightarrow \emptyset(T)_{o\epsilon\Sigma}$, where each package name $p_i \in \Sigma$ in the vector is assigned a binary flag depending on its appearance (one) or not (zero) in the trace T. The term vector can be weighed by the term frequency (*tf*):

$$\phi_{tf}(p,T) = freq(p_i); i = 1, \ldots, m \quad (2)$$

where *freq(pi)* is the number of times the package p_i appears in T normalized by *L* (the total number of package calls in *T*).

The term frequency considers all terms as equally important across all documents or collection of traces (Γ). However, rare terms that frequently appear in a small number of documents convey more information than those that are frequent in most documents. The inverse document frequency (IDF) is proposed to increase (or decrease) the weights of terms that are rare (or common) across all documents. The term vector weighed by the TF-IDF is therefore given by:

$$\phi_{tf.idf}(p, T, \Gamma) = \frac{\mathrm{K}}{df(p_i)} freq(p_i); i = 1, \dots, m \quad (3)$$

where the document frequency $df(p_i)$ is the number of traces T_k in the collection of Γ of size *K* that contains the package name p_i. A high weight in TF-IDF is thereby given to package names that are frequent in a particular trace $T \in \Gamma$, but appear in few or no other traces of the collection, Γ.

The weighing strategy of Equation (3) discards the temporal order of packages. The proposed approach, DURFEX, however, accounts for the temporal order of package occurrences by extracting and mapping variable length N-grams and their frequencies from each trace $T \in \Gamma$, to fixed-size feature vectors. Each N-gram is a sequence of contiguous package names of length N extracted from trace *T*.

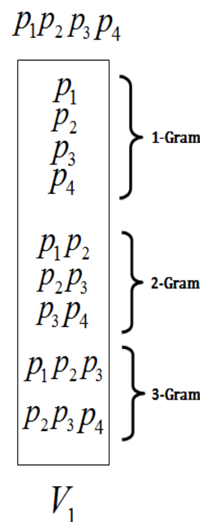

Figure 14 Variable length N-gram

As illustrated in Figure 14, the feature extraction starts by sliding a window of N package names over the trace $T \in \Gamma$, shifted by one package name. For each sequence, the individual (or 1-gram) package names are first extracted, followed by all N-grams for n = 1,…,N that are rooted at the first package name inside the sliding window, which are then organized in vectors *Vi* (see Figure 14).

With the arrival of each new incoming bug report B_i, a dictionary of all variable length N-grams from the packages extracted from vectors $\Gamma = T_1, T_2, \ldots, T_K$ is constructed. The constructed dictionary has a distinct alphabet Σ with the size $m = |\Sigma|$. Each feature in the feature vector is weighed according to the frequency and inverse document frequency of that feature in N-gram features extracted from each $T \in \Gamma$, corresponding to the stack traces in the bug reports.

The unique N-grams (for n = 1,…,N) from all unique vectors *Vi* obtained by sliding the window over the available traces $T \in \Gamma$ are used as dictionary keys, while their accumulated frequencies are used as values. This dictionary of size, say *D*, is, therefore, the reference database representing the 1-grams, 2-grams, . . . , N-grams that occurred in the collection of traces Γ. Finally, each vector *Vi* (shown in Figure 14) is mapped to the space *D* of the reference dictionary and becomes a feature vector.

For a specific process, the size of the dictionary (*D*), which is the size of the unique feature vectors, depends on the alphabet size *m*, the sliding window size *N* and the regularity of the process. The value of *N* is a user-defined parameter that influences the detection power and the size of the feature vectors. A small *N* value is always desirable since it results in smaller feature vectors, and hence allows faster detection and response during operation.

For each $T \in \Gamma$, if we assume $|T|=L$, then the maximum number of N-gram features or sequences of *N* package names that can be extracted is L– (Ngram - 1). For example, the number of 3-gram features that can be extracted from a sequence of L packages is L-2. Considering that varying N-gram is the collection of features extracted by applying 1-gram, 2-gram up to N-gram, the maximum number of features that can be extracted from $|T|=L$ denoted as F can be calculated as Equation (4).

$$F= L + L -1 + L-2 +....+ L – (N\text{-gram} - 1)$$

$$F = N * L - \frac{N*(N-1)}{2} \qquad (4)$$

In practice, given that one package can appear more than once and overlaps of N-grams are common, the number of features is far less than what is stated in Equation (4). According to Equation (4), the number of features extracted from *|T |=L* is always less than the number of grams multiplied by the length of the sequence of packages in the stack traces. In our case studies, we found that the most suitable number of grams is at most three. This shows that even by using varying length N-grams to extract feature vectors from packages, the number of features remains by far less than the number of unique functions.

The only non-null elements of the feature vectors are those that correspond to the N-grams of the original vector (*Vi*) before the mapping. These elements are weighed by their frequencies and inverse document frequencies that have been accumulated as values in the reference dictionary. We conducted experiments using TF-IDF.

5.2. Proposed Approach

We conducted an investigation on groups of duplicate bug reports in eclipse. We noticed that in most cases, not only the stack traces but also the categorical information such as severity and component is the same for all duplicate bug reports in the same group. Thus, we extended our similarity function to incorporate these categorical features as well. In this chapter, we define the similarity function between two bug reports (B_1, B_2) as follows:

$$SIM\ (B_1, B_2) = \sum_{i=1}^{3} w_i * feature_i \ (5)$$

Where $feature_1$, $feature_2$ and $feature_3$ are defined as follows:

$$feature_1 = similarity\ of\ abstracted\ stack\ traces$$

$$feature_2 = \begin{cases} 1, if\ B1.components = B2.Component \\ 0, \qquad otherwise \end{cases}$$

$$feature_3 = \begin{cases} 1, if\ B1.seveirty = B2.severity \\ 0, \qquad otherwise \end{cases}$$

According to Equation (5), the similarity of two bug reports is linear combination of their stack traces and categorical features similarity.

The distance between two stack traces is measured as the distance between their corresponding feature vectors, weighed according to Section 5.1. We use cosine similarity to measure the similarity between two vectors.

Typically, given $V_1 =< w_{11}, w_{12}, \ldots .. w_{1n} >$ and $V_2 =< w_{21}, w_{22}, \ldots .. w_{2n}$, the cosine similarity is calculated using Equation (6) [MRS08]:

$$Cos(\theta) = \frac{V_1.V_2}{|V_1|.|V_2|} \quad (6)$$

The cosine similarity between two vectors, which is the cosine of the angle between two vectors, is equal to the dot product of the two vectors divided by the multiplication of their sizes.

The SIM function contains three free parameters (w_1, w_2, w_3) which must be optimized. We used 10% of our dataset for training these parameters. Our training dataset format is consistent with Sun et al. [BJSWPZ08]. It contains triples in the form of (*q, rel, irr*) where *q* is the query bug report, *rel* is a duplicate bug report and *irr* is a bug report which is not duplicate of *q*. The method used to create the training set is depicted in Figure 15 [BJSWPZ08].

```
TS = Ø: training set
N>0 size of TS
G: Group of duplicate bug reports
----------------------------------
For each Group G in the repository do
  R= all bug reports in Group G
  For each report q in R do
     For each report rel  in R-{q} do
        For i=1 to N do
            Randomly choose a report irr out of R
            TS=TS ∪ {(q,rel,irr)}
        End for
     End for
  End For
End for
Return TS
```

Figure 15 Training Dataset

We need to define a cost function to optimize the free parameters (w_1, w_2, w_3) based on our training set. We define the RankNet cost function as follows [BJSWPZ08]:

$$Y = Sim(irr,q) - sim(rel,q)$$

$$RNC(I) = Log(1 + e^{Y}) \quad (7)$$

The cost function is minimized when the similarity of a bug report and its duplicate is maximized and the similarity of a bug report and its not a duplicate bug report is minimized. We used gradient descent as shown in Figure 16 [BJSWPZ08] to minimize the above cost function. The optimization algorithm adjusts each free parameter *x* in each iteration according to coefficient η and partial derivative of *RNC* with respect to each free parameter *x* [BJSWPZ08].

TS = ∅*: training set* *N>0 size of TS* η*: the tuning rate*

For *n=1 to N* **do**
   For each instance *I* in *TS in random order* **do**
      For each free parameter x in *sim()* **do**
         x=x- $\eta * \frac{\partial RNC}{\partial x}$
      End for
   End for
End For

Figure 16 Optimization using gradient descent

We optimize the free parameters of the linear model of Equation (5) on the dataset created based on Figure 15, using 10% of the dataset. After training the model, we can extract the weights w_1, w_2, w_3. These extracted weights are then used to detect duplicate bug reports online. For the rest of the dataset, with the incoming of each bug report its stack trace similarity and categorical features similarity to all the previous bug reports is calculated. Stack trace similarity is obtained by comparing stack traces weighed using term frequency and inverse document frequency. Categorical features similarity is one if they have the exact same value. It is zero otherwise. Next, we use the w_1, w_2, w_3 weights to calculate the similarity of bug reports using the linear combination of the stack traces and categorical features. After calculating the similarity of bug reports to all the previous bug reports in the dataset, a list of *N* most similar bug reports is returned as the potential duplicates of the incoming bug reports. In the next section, we explain

the evaluation metric, which is used for evaluating the accuracy of the proposed approach in detecting duplicate bug reports.

DURFEX is as an online duplicate detection approach. Figure 17 shows an overview of how DURFEX is applied.

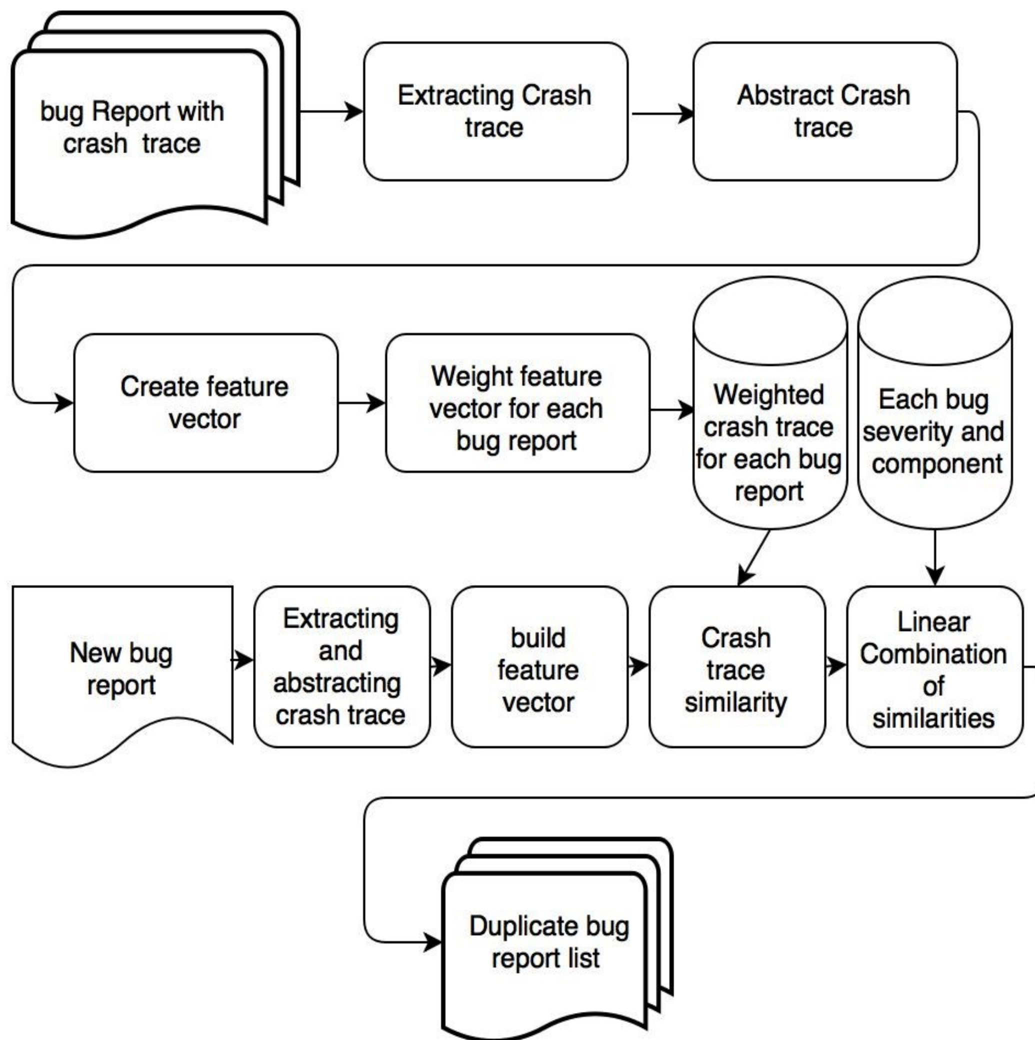

Figure 17 Proposed approach (DURFEX)

5.3. Evaluation

The objective of the experiment is to investigate if stack traces represented as package names instead of the raw trace of function calls can be used to detect duplicate bug reports, and if so, what would be the accuracy? More precisely, the experiment aims to answer the following questions:

RQ1. Can traces of package names alone be used to detect duplicate bug reports?

RQ2. What is the accuracy of detecting duplicate bug reports using traces of package names and categorical features compared to traces of function calls?

RQ3. What is the processing time of DURFEX compared to the approach which uses stack traces only?

5.3.1. The Dataset

The dataset used for evaluation consists of bug reports of the Eclipse bug repository. In this repository, stack traces are embedded in the bug report description. To extract the content of the stack traces in Eclipse, we use the same regular expression presented by Lerch et al. [LM13]. For Eclipse, the extracted traces are preprocessed to remove the noise in the data such as native methods, which are used when a call to the Java library is performed. There are also lines in the traces that are labelled 'unknown source'. This occurs due to the way debugging parameters are set. We removed those lines from stack traces too.

Table 3 characteristics of the dataset

Data	Eclipse
Total number of reports	455,700
Total number of bug reports	388,827
Total number of enhancement reports	66,873
Total number of bug reports with stack traces	42,853
Total number of duplicate bug reports with stack traces	8,834
Total number of groups of duplicate bug reports with stack trace	3,278

The dataset contains Eclipse bug reports from October 2001 to February 2015. This dataset contains 455,700 reports, of which 388,827 are bugs and 66,873 are labelled as enhancement (see Table 3). Among all these bugs, 42,853 had one or more stack traces in their description and, if labelled as duplicate, can be used to assess duplicate detection capability of our approach. We have a total of 8,834 duplicate bug reports that are distributed in 3,278 groups which contain at least two duplicates.

5.3.2. Dataset Analysis

In Eclipse, almost 68% of duplicate bug report groups contain only two bug reports, while the other 32% are distributed among groups of 3 or more (see Figure 18). Pursuing this further, the fact that the duplicate bug report groups rarely contain more than six duplicate bug reports shows that the problem of detecting duplicate bug reports in the Eclipse bug repository is challenging. Most of the time, there exists only one duplicate bug report for each bug report in the repository.

To compare the difference when using package names instead of functions, we measure the total number of unique functions and packages. The result is shown in Figure 19. It shows the number of unique functions and unique packages in the Eclipse dataset between 2002 and 2014. For the bug reports submitted in 2002, the number of functions is 7,670 compared to 534 packages (which represents only 7% of the number of functions). The bug reports submitted until 2014 contain 130,483 unique functions defined in only 9,733 packages (7.4%). Clearly, the use of package names instead of functions is a suitable abstraction level for detecting duplicate bug reports size-wise.

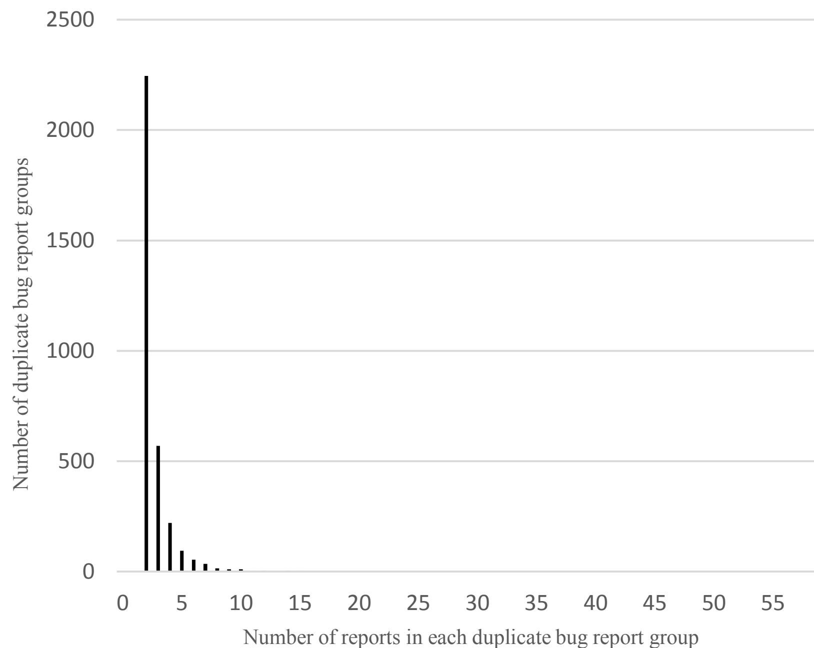

Figure 18 The number of duplicate reports in all duplicate bug reports groups in Eclipse

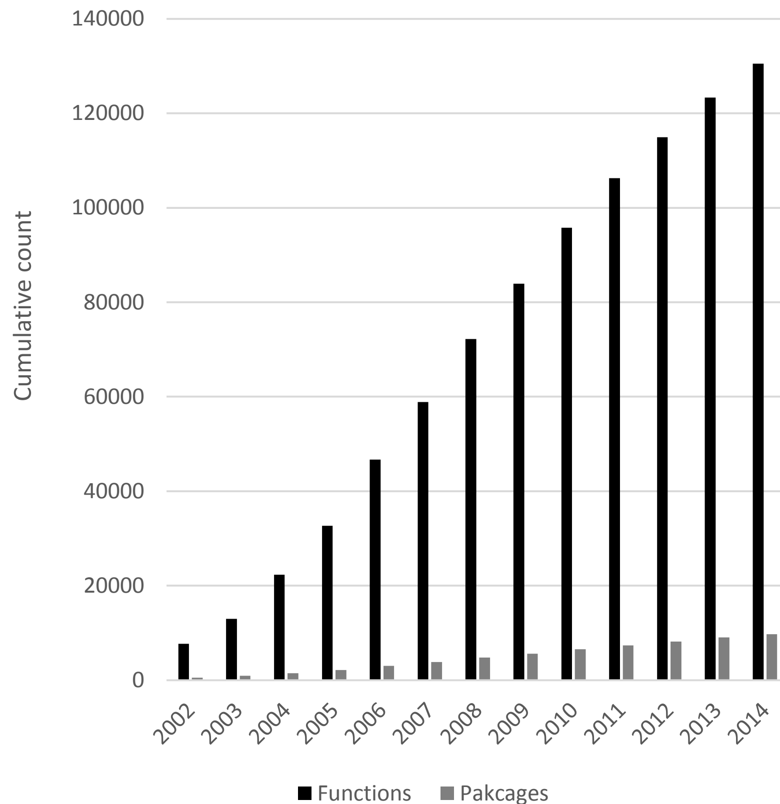

Figure 19 Cumulative number of distinct functions and packages in Eclipse

We also studied the period in which most of the duplicate reports appear. This can help design an approach that considers only the periods where most of the duplicate reports are submitted. For example, Runeson et al. [RAN07] conducted a study on the Sony-Ericsson bug repository and concluded that a time window of 50 days has the best recall rate for the Ericsson dataset. Their technique uses natural language processing of bug report descriptions.

In the Eclipse dataset, the period in which the duplicate bug reports are submitted is shown in Figure 20. In this figure, the time between the creation of the first and last bug report in the group is computed. Most duplicate bugs are reported within a time window of 100 days. Almost 81% of bug reports and their duplicates can be placed in a time window of 100 days. The last bar in Figure 20 shows the number of bug reports which their distance to their duplicate is farther than 500 days.

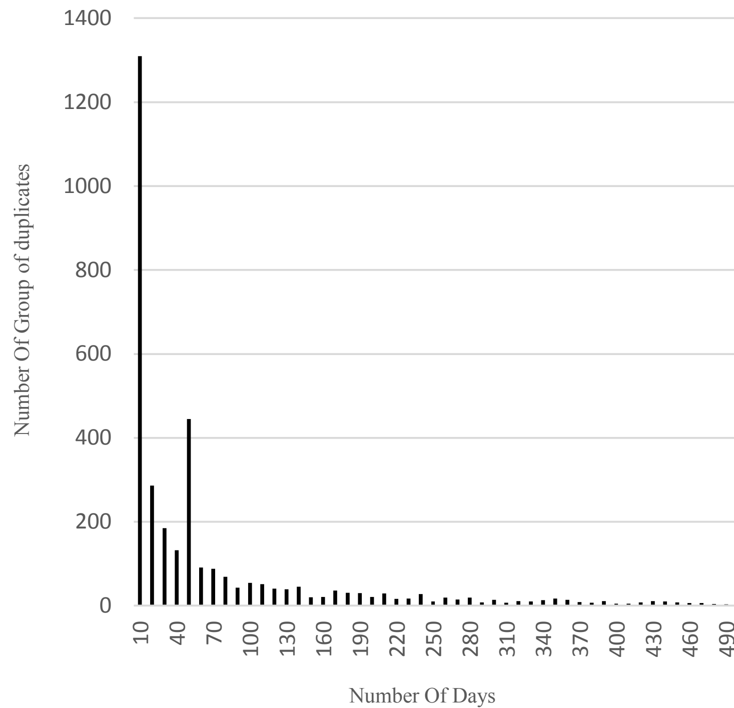

Figure 20 Number of days between the first and last bug report in the duplicate group in Eclipse

5.3.3. Evaluation Measure

Precision and recall are popular evaluation metrics for classification problems. In this study, since we seek to find the duplicate of a given bug report among existing bug reports, precision and recall do not seem to be suitable metrics. Instead, similar to Runeson et al. [RAN07] and Lerch et al. [LM13], in this chapter, the duplicate bug report detection accuracy is measured using the recall rate and mean reciprocal rank.

Recall rate is defined as the percentage of duplicate bug reports from the whole duplicate bug reports set in our system, for which their duplicate bug report was found by the approach in the suggested list (Equation (1)).

The list size represents the number of bug reports that are most similar to the incoming report. This is the maximum number of bug reports that the triager needs to examine to find the duplicate bug report. The chance of finding the duplicate bug report increases by increasing the size of the list. For example, for the list size of 10, for each incoming bug report a list of the ten most similar bug reports (potential duplicates) is constructed. If the duplicate of the incoming bug report is among these ten bug reports, it

is counted as a detected duplicate. The recall rate is then defined as the portion of bug reports for which their duplicate is detected, given a certain list size (Equation (1)).

The recall rate, however, does not show the position of the duplicate bug report in the suggested list. To evaluate the accuracy of the approach based on the rank of duplicate bug reports in the suggested list, Lerch et al. [LM13] introduced the mean reciprocal rank (MRR), which is defined as:

$$MRR = \frac{1}{Q}\sum_{i=1}^{|Q|}\frac{1}{rank_i} \quad (8)$$

In MRR, Q is defined as the number of bug reports, which have at least one duplicate in the list and $rank_i$ is the rank of the duplicate bug report in the suggested list. MRR approaching one means that the duplicate is ranked among the first reports. MRR approaching to zero means that the detected duplicate is placed on the bottom of the list.

5.4. Results and Discussion

In this Section, we discuss the results of applying DURFEX to stack traces and categorical features of the bug reports of the Eclipse bug repository. We also compare our approach to the one presented by Lerch et al. [LM13], where the authors used only function names with the term frequency and inverse document frequency as the weighing mechanism. Consistent to Lerch et al. [LM13], when using text to detect duplicate bug reports, we used bug report descriptions, including stack traces.

5.4.1. Comparison of DURFEX to the Use of Function Calls

Figure 21 shows that in the Eclipse dataset, using package names (only 1-gram) (weighed by their frequencies of occurrence and inverse document frequency) and categorical features, we have a recall rate of up to 84%. This result answers RQ1 and shows that packages and categorical features could be used to detect duplicate bug reports.

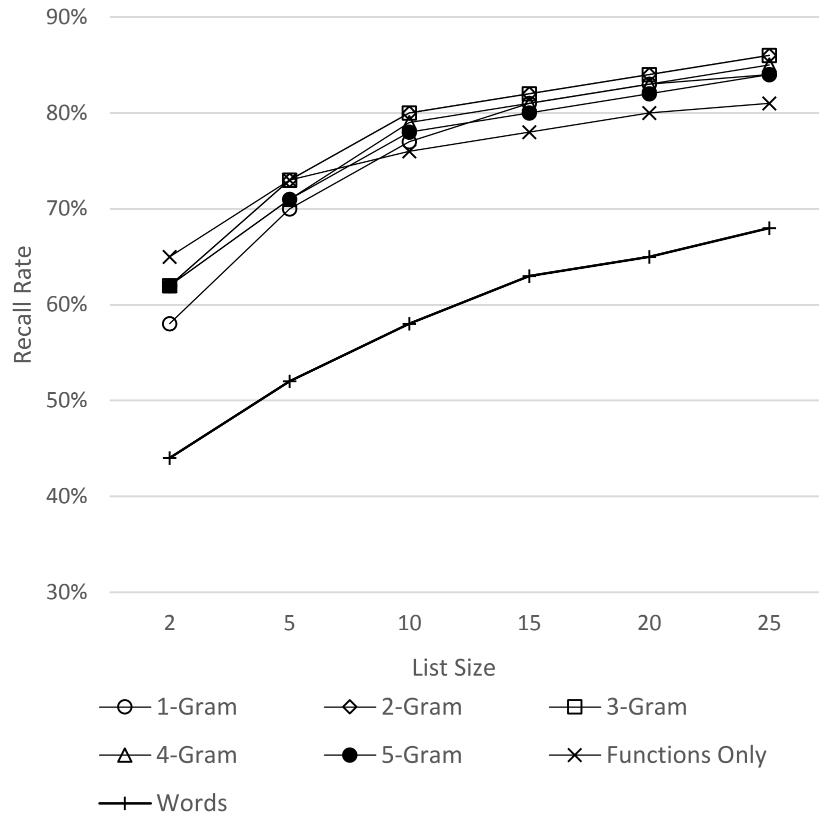

Figure 21 Recall rate of DURFEX with different N-grams compared to the use of unique functions in Eclipse dataset

Table 4 DURFEX recall rate on the Eclipse dataset

Approach	List Size					
	2	5	10	15	20	25
Packages (1-gram)	58%	70%	77%	81%	83%	84%
Packages (2-gram)	62%	73%	80%	82%	84%	**86%**
Packages (3-gram)	62%	73%	80%	82%	84%	86%
Packages (4-gram)	62%	71%	79%	81%	83%	85%
Packages (5-gram)	62%	71%	78%	80%	82%	84%
Functions Only	65%	73%	76%	78%	80%	**81%**
Words	44%	52%	58%	63%	65%	68%

In Figure 21, the recall rate is best (86%) when using 2-gram sequences, which outperforms using distinct function names only (81%) see Table 4. Mean reciprocal rank (Table 5) is almost the same when using

1-gram packages and categorical features compared to functions. Using 2-gram, our approach outperforms using function calls only. In addition, in all cases using abstracted stack traces outperforms using bug reports textual descriptions. This is consistent with the result presented by Lerch et al. in [LM13]. Based on the results presented in Figure 21, Table 4 and Table 5 we can answer RQ2 and conclude that using packages and categorical features, we are not only able to significantly decrease the number of features (which yields lower processing time), but also improving the performance of the approach.

Table 5 DURFEX Mean reciprocal rank on the Eclipse dataset

Approach	Eclipse without a time window
DUEFEX (1-gram)	65%
DUEFEX (2-gram)	68%
DUEFEX (3-gram)	68%
DUEFEX (4-gram)	68%
DUEFEX (5-gram)	68%
Function calls	64%
Bug report descriptions	58%

5.4.2. Comparison of DURFEX to the Use of Function Calls with a Time Window

As we discussed earlier, the number of days between the submission of a bug report and its duplicate is less than 100 days in 81% of cases in the Eclipse dataset. This result encouraged us to study the effect of having a time window when detecting duplicate bug reports. This was also used by Runeson et al. [RAN07].

Figure 22 (Table 6) shows the results of applying DURFEX to the Eclipse dataset comparing bug reports that appear only in the time window of 100 days before the submission of a new bug report. The recall rate obtained by DURFEX with 2-gram and 3-gram is 89%, which is better than the results obtained when using functions only. In Table 7, the mean reciprocal rank is best when using 2-gram (73%) which outperforms using distinct functions only (59%) and using bug report description.

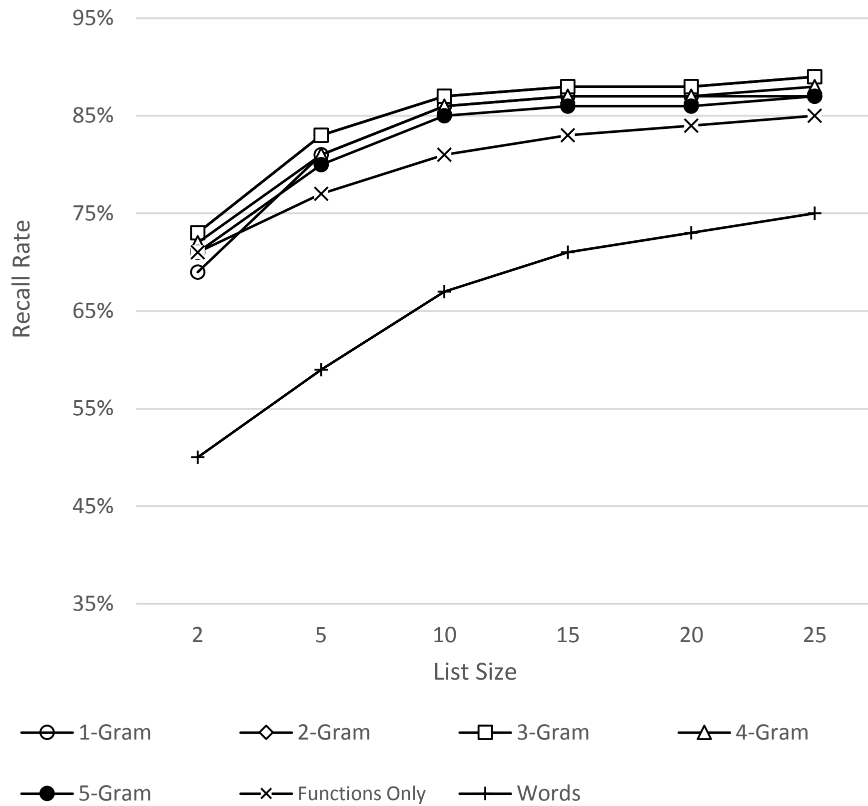

Figure 22 Recall rate of DURFEX with different N-grams compared to the use of distinct functions in the 100-day-time period in Eclipse

Table 6 DURFEX results on the bug reports of the 100-day period on Eclipse

Approach	List Size					
	2	5	10	15	20	25
Packages (1-gram)	69%	81%	86%	87%	87%	87%
Packages (2-gram)	73%	84%	87%	88%	88%	**89%**
Packages (3-gram)	73%	83%	87%	88%	88%	**89%**
Packages (4-gram)	72%	81%	86%	87%	87%	88%
Packages (5-gram)	71%	80%	85%	86%	86%	87%
Functions Only	71%	77%	81%	83%	84%	85%
Words	55%	65%	72%	76%	78%	80%

These findings are very promising since they suggest that we can detect duplicate bug reports using an abstract trace (in our case, a trace of packages) instead of raw traces (this answers RQ1) with better

accuracy (this answers RQ2). In the next subsection, we will show that the gain in terms of the execution time is huge. DURFEX is considerably more efficient than an approach that uses functions only.

Table 7 DURFEX Mean reciprocal rank on the bug reports of the 100-day period on Eclipse

Approach	Eclipse without a time window
DUEFEX (1-gram)	70%
DUEFEX (2-gram)	73%
DUEFEX (3-gram)	73%
DUEFEX (4-gram)	73%
DUEFEX (5-gram)	73%
Function calls	59%
Bug report descriptions	58%

5.4.3. Comparison of Processing Time Using Functions and Packages

We measured the time it takes to find a duplicate bug report using DURFEX and compare it to the time it takes to find a duplicate report using distinct functions in stack traces to show the performance of DURFEX compared to an approach that only uses function calls such as the one from Lerch et al. [LM13]. Table 8 shows the average execution time when applying DURFEX to Eclipse dataset (2002-2014), compared to an approach that uses functions (Figure 23 shows detailed results for each year of the dataset for Eclipse). As we can see, with 1-gram, DURFEX execution time represents only 7% of the execution time of the approach that uses functions as features. This is a gain of 93%. When used with 2-gram (that provide the best detection accuracy as discussed in the previous Section), DURFEX execution time represents 32% of the execution time of the approach that uses functions. In other words, DURFEX (with 2 gram) runs on average, around 70% faster than an approach that uses functions only (this addresses the research question RQ3). Finally, we want to note that when DURFEX is used with 4 and 5 grams, we start noticing a decrease in the processing time gap. This is expected since the larger the N-gram, the more features we have, which results in a longer time in processing duplicate bug reports.

Combined with the results shown in the previous subsection, we demonstrated that abstract traces of package names can be used to detect bug reports with good accuracy while keeping the execution time considerably low. We believe that this makes DURFEX a practical approach for detecting duplicate bug reports.

Table 8 Comparing the average execution time in seconds of DURFEX to the execution time of an approach that uses function calls applied in ECLIPSE. The percentage indicates the ratio of the execution time of DURFEX to the execution time taken by an approach that uses function calls only

Approach	Eclipse
Use of function calls	5660.71
DURFEX (1-gram)	400.6 (7%)
DURFEX (2-gram)	1866.42 (32%)
DURFEX (3-gram)	4511.89 (79%)
DURFEX (4-gram)	8382.42 (148%)
DURFEX (5-gram)	13442 (237%)

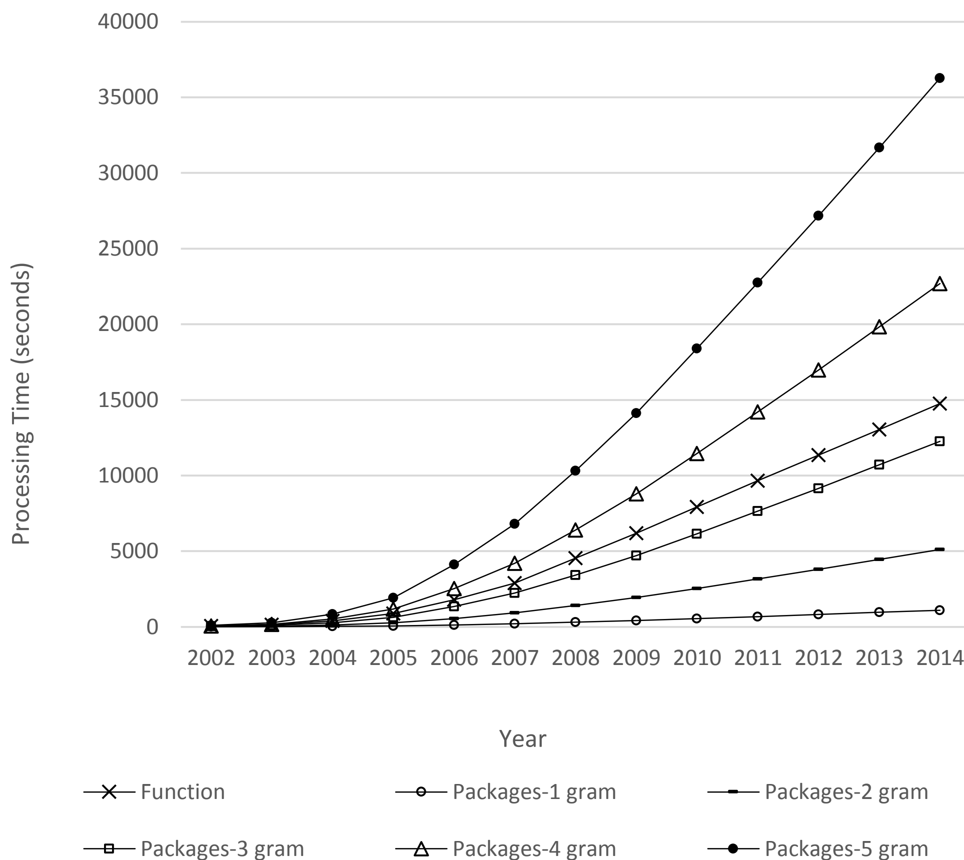

Figure 23 Comparison of processing time of in Eclipse dataset

5.5. Threats to Validity

The regular expression used may miss some stack traces causing false negatives. Using a more robust parser or a better regular expression may affect the number of available stack traces and thus may change the recall rate.

In our dataset, the completeness of stack traces can be a threat to validity. Since stack traces are copied by users who are not necessarily experienced, only a section of the traces that seems to be important may have been copied by users in the description of the bug report. Incomplete stack traces may have affected the accuracy of the proposed approach.

In this chapter, we used the Eclipse dataset. To generalize our approach, we need to apply it to more datasets (both public and proprietary).

5.6. Conclusion

In this chapter, we proposed an approach, called DURFEX that, if implemented in bug tracking systems, can facilitate the process of bug handling by automatically detecting duplicate reports. Our approach is based on the concept of trace abstraction, where we turn stack traces of function calls to traces of packages (by replacing each function with its containing package). DURFEX keeps track of the temporal order of calls by creating feature vectors from package name traces using varying length N-grams.

When applied to the Eclipse bug repository, our approach achieves better results than an approach that uses function calls. The difference is that the execution time of DURFEX can be 93% less than an approach that uses functions only with the 1-gram, approximately up to 70% less using 2-gram. These results demonstrate the effectiveness of DURFEX compared to an approach that uses functions only.

Chapter 6

Bug Severity Prediction

The main objective of bug tracking systems is to enable end-users to submit bug reports, where they can report various information about the bug. This information includes the textual description, the stack trace, and other categorical information, such as the perceived severity of the bug and its operating environment (e.g., product, component and operating system). This information is then used by developers to fix the bug.

After a bug report is submitted, a team of triagers examine each report in order to redirect the ones requiring fixes to the developers of the system. For software systems with a large client base like Eclipse, triaging is a tedious and time-consuming task. To balance the workload of the developers and optimize the bug handling process, not all the bug reports are handled at the same time. Bug reports need to be prioritized. This is due to the large number of reports received [DMJ11]. Triagers usually prioritize the bug reports using the reported bug severity typically. Bug severity is defined as a measure of how a defect affects the normal functionality of the system [LDSV11, YHKC12]. It is used as an indicator of how soon a bug needs to be fixed [ZYLC15].

In theory, a user chooses the severity of a bug according to its defective appearance. However, despite the existence of guidelines on how to determine the severity level of a bug, studies show that the severity level is often incorrectly assigned by the users [ZCYLL16]. As a result, the severity that is submitted in the bug report does not reflect the real impact of the bug on the system, which affects the bug report handling process. This is mainly due to the fact that users are not expert in the system domain, so they assign incorrect severity levels to the bugs. An inaccurate severity level causes a delay in the processing of the bug reports [ZCYLL16].

To address this issue, several prediction methods [LDSV11, AADPKG08, MM08] have been developed to predict the correct severity level of a bug. These techniques treat the problem as a classification problem by learning from historical bug reports in order to classify the incoming ones. They use mainly the words in the bug report descriptions as the main features for classification.

In this chapter, we propose two new bug severity prediction techniques. An approach based on stack traces and another approach based on stack traces and categorical features. We show that stack traces could be used to predict the severity of bugs more accurately than bug reports descriptions. We also show that adding categorical information to the stack traces further improves the severity prediction accuracy. To the best of our knowledge, this is the first time that stack traces are used to predict the severity of bugs. We evaluate our approach on 11,825 and 153,343 bug reports from Eclipse and Gnome bug repositories respectively.

This chapter is organized as follows: We discuss the features used for severity prediction in Section 6.1. The Severity labels of Gnome and Eclipse dataset is explained in Section 6.2. The proposed approaches are explained in Section 6.3. We discuss the dataset, evaluation approach and results in Section 6.4. Threats to validity are presented in Section 6.5. The conclusion is presented in Section 6.6.

6.1. Bug Report Features

A bug report consists of the bug description, the bug stack trace and other categorical information. Bug report description provides information about bug report using natural text, and stack trace provides more information about the execution path of the software when it crashed. Categorical information in the bug report provides information about the environment in which the bug has been discovered. Such information is important when assigning a bug severity, particularly for system and integration bugs, which are the most difficult ones to identify at the development time [MM08].

In this chapter, we choose to use three categorical features to predict the severity of a bug. These three categorical features include the product, the component, and the operating system of the bug report.

Each bug report categorical field will be assessed by the triager after being submitted to the bug tracking system. The assessed bug report fields, if needed, will be further adjusted by the triagers who are experts in the software system.

The severity of a bug is defined as a measure of how much the bug affects the functionality of a software system [LDSV11, YHKC12]. If a bug stops the main functionality of the software system (e.g. the Eclipse IDE editor stops responding) then we can categorize that as a high severity bug. Eclipse and Gnome are both platforms which are composed of different products. We chose product as one of the categorical features because bugs that are bounded to products which are crucial for the main functionality of the software system could be an indicator of high severity bugs. Each product is composed of a set of components. Components are used in bug reports to locate defective locations more precisely inside each product. Since not all components of a product, which controls important functionality of a software system, are of the same importance to the functionality of the product, we also use components as another categorical feature for predicting the severity of bugs. Furthermore, when a software system runs on different operating systems, different types of functionalities could be expected. Software behaviour is usually related to the operating system it is being run on. We expect the similarity of bugs caused by running software on the same operating systems to be a better indicator for severity prediction. We did not include the version of the software, since a bug may exist in multiple versions of the software and not be version specific. A bug can be reported by many different users using different versions while having the same underlying reason and severity.

6.2. Levels of Bug Severity

Severity can be manually assigned by a user and describes the level of the impact of a bug. The followings are the severity levels that can be found in Eclipse and Gnome bug tracking systems:

- Blocker[4] severity shows bugs that halt the development process. Blocking bugs are the bugs that do not have any workaround.
- Critical[4] bugs are the bugs that cause loss of data or sever memory leak.
- Major[4] bugs are the bugs that are a serious obstacle to work with the software system.
- Normal[4] bugs are advised to be chosen when the user is not sure about the bug or if the bug is related to documentation.
- Minor[4] bugs are the ones that are worth reporting but do not interfere with the functionality of the program.
- Trivial[4] bugs are cosmetic bugs.
- Enhancements[4] bugs are the gray areas, including new features that are not considered a bug.

There are two types of severity prediction techniques based on the level of granularity of the severity labels (coarse-grained and fine-grained severity prediction). In the literature, many studies focused on coarse-grained severity prediction, in which, the severity of a bug takes two values: non-severe or severe. In these studies [LDSV11, LDGG10], severe bugs are bugs with Blocker, Critical or Major severity, while non-severe bugs are bugs with Minor or Trivial severity [LDSV11]. Non-severe bugs can stay in a bug routing queue for a longer period of time, while severe bugs must be routed immediately to the developers to provide fixes. In fine-grained severity prediction [TLS12], each of the severity labels is considered separately and is not abstracted out to non-severe and severe categories. This helps bug triagers to optimize resources more efficiently. In this chapter, we focus on fine grain severity prediction approaches.

[4] https://wiki.eclipse.org

6.3. The Proposed Approach

In this Section, we propose a new severity prediction approach that uses stack traces to predict the severity of bugs. We also propose an approach, which predicts the severity of bugs using the combination of stack trace and categorical information. We predict severity by calculating the similarity of each incoming bug report to all the previous bug reports in the bug tracking system. The similarity of bug reports is calculated based on the linear combination of the similarity of their stack traces and the similarity of their categorical features including product, component and operating system. The severity of the bug report is then selected based on the severity of the K nearest neighbour (K nearest bug reports) to the incoming bug.

6.3.1. Predicting the Bug Severity Using Stack Traces and Categorical Features

We follow an approach based on the linear combinations of stack traces and categorical features similarity to calculate the similarity of bug reports and predict the severity of bugs based on stack traces and categorical features. Given two bug reports (B_1, B_2) the combined similarity is calculated as follows:

$$\text{SIM}\ (B_1, B_2) = \sum_{i=1}^{4} w_i * feature_i \quad (9)$$

Where $feature_1, feature_2$, $feature_3$ and $feature_4$ are defined as follows:

$$feature_1 = Similarity\ of\ stack\ traces$$

$$feature_2 = \begin{cases} 1 & if\ B1.Product = B2.Product \\ 0 & otherwise \end{cases}$$

$$feature_3 = \begin{cases} 1 & if\ B1.Component = B2.Component \\ 0 & otherwise \end{cases}$$

$$feature_4 = \begin{cases} 1 & if\ B1.operating\ system = B2.operating\ system \\ 0 & otherwise \end{cases}$$

Based on Equation (9), the similarity of two bug reports is the linear combination of their corresponding stack traces and categorical features similarities. In Equation (9) similarity of categorical features is one if they are the same and zero if they are not. The approach which predicts severity based on stack traces only is explained in Section 6.3.2. The method used to extract features (stack trace and categorical) is

explained in Section 6.3.3 , and the method used for calculating the similarity of stack traces is explained in Section 6.3.4. The K nearest neighbour method used for predicting severity is explained in Section 6.3.5. The introduced similarity function in Equation (9) has four tuneable parameters w_i: (w_1, w_2, w_3, w_4), which represent the weights of each feature. These four parameters must be optimized to reflect the importance of each feature. To tune these parameters, we used an adaptive learning approach that uses a cost function and gradient descent in a similar way to the one used by Sun et al. [SLKJ11]. The format of our training set is similar to the one used by Sun et al. [SLKJ11]. The dataset contains triples in the form of (q, rel, irr) where *q* is the incoming bug report, *rel* is a bug report with the same severity and *irr* is a bug report with a different severity. The method used to create the training set (TS) is shown in Figure 24.

```
TS = ∅: training set
N>0: size of TS
G: Group of bug reports with the same severity
-----------------------------------------------
For each Group G in the repository do
  R= all bug reports in Group G
  For each report q in R do
     For each report rel  in R-{q} do
        For i=1 to N do
            Randomly choose a report irr out of R
           TS=TS ∪ {(q,rel,irr)}
        End for
     End for
  End For
End for
Return TS
```

Figure 24 Training Dataset

Based on Figure 24, at the first step bug reports are grouped based on the severities. Next, for each bug report (*q*) in each severity group, we need to find bug reports which have the same severity and bug reports with other severities. We chose a bug report from the same severity group (*rel*) and another bug report from another severity group (*irr*). The process continues till we create all (*rel*, *irr*) pairs for the bug report (*q*). We continue the same steps for all the bug reports in our training set to create the (*q, rel, irr*) triples.

As explained earlier, to optimize the free parameters (w_1, w_2, w_3, w_4), we need to define a cost function based on the created triple format. The selected RankNet cost function [SLKJ11] is shown in (7). Our goal is to minimize the defined RankNet cost function. The cost function is minimized when the similarity of bug reports with the same severity (defined in Equation (9)) is maximized, and the similarity of the bug reports with different severities is minimized. To minimize the above cost function, we used a gradient descent algorithm as shown in Figure 16.

6.3.2. Predicting Severity Using Stack Traces

To predict severity using stack traces only, the similarity of bug reports is defined as the similarity of their stack traces only. When predicting the severity of bugs using stack traces and categorical features, we needed a separate training step to tune the importance (weight) of similarity of stack traces and categorical features. However, since the approach which uses stack traces to predict severity of bugs considers the similarity of bugs as similarity of their stack traces, we do not have tunable parameters and we do not need separate training phase for tuning those parameters.

6.3.3. Bug Features Extraction (Stack Traces and Categorical Features)

In Bugzilla, users copy the stack trace of a bug inside the description of the bug report. To identify and extract these traces, we use the regular expression of Figure 5 of Section 3.1 [LM13]. Moreover, to extract stack traces form the Gnome bug repository, we defined and implemented the regular expression presented in Figure 6 of Section 3.2. Normally, categorical features can be found in the XML preview of a bug report. Different categories use different XML tags that conform to the Bugzilla schema. In our work, we implemented a custom parser to extract this information from the bug reports XML previews of Eclipse and Gnome bug tracking system.

6.3.4. Stack Trace Similarity

For each incoming bug report, its stack trace and the stack traces of all bug reports previous to that in the bug tracking system are extracted. We create feature vector from all distinct functions from the extracted stack traces, then we weigh the feature vector for each stack trace using term frequency and inverse document frequency of Equation (3) and compare the weighed feature vectors together using cosine similarity of Equation (6).

6.3.5. KNN Classifier for Severity Classification

After calculating the similarity of bug reports using either the linear combination of their stack traces and categorical feature similarity or their stack traces similarity only, we use KNN to predict bug report severity. KNN is an instance-based lazy learning algorithm [TLS12]. Given a feature vector, KNN returns the K most similar instances to that vector. Following a typical KNN classification algorithm, in our case, KNN has two phases: In the first phase, the similarity of the incoming bug report B_i to all the bug reports in the training set is calculated. Accordingly, based on the value of (*k*), which is a constant value that defines the number of returned neighbours, the algorithm returns the *K* nearest relevant instances. In the second phase, a voting algorithm (e.g., majority voting) among the labels of the *k* most similar instances is used to classify the incoming bug report B_i. The label of instance *X* (i.e., the instance that corresponds to B_i) can be determined given the labels set C by majority voting as in Equation (10) [PF13].

$$\mathrm{C(X)} = \underset{c_j \in C}{argmax} \quad score\ (c_j, neighbors_k(X)) \quad (10)$$

In Equation (10) $neighbors_k(X)$ is the *K* nearest neighbours of instance *X*, argmax returns the label that maximizes the score function, which is defined in Equation (11) [PF13].

$$Score\ (c_j, N) = \sum_{Y \in N} [class(y) = c_j] \quad (11)$$

In Equation (11), $class(y) = c_j$ returns either one or zero. It is one when $class(y) = c_j$ and zero otherwise. In Equation (11), the frequency of appearance of a label is the only factor that determines the

output label. Finally, the label with the highest frequency among K labels is chosen as the label of the incoming bug report.

While the score function in Equation (11) shows promising results, to further enhance the prediction capability of our approach, we also need to consider the similarity of each top *K* returned bug reports by giving more weights to the label of the bug reports that are closer to the incoming bug report. The reason is that, bug reports that are closer to the incoming bug report B_i will most probably have the same label as B_i.

If we assume the distance of the closest bug report in the sorted list as $dist_1$ and the distance of the farthest bug report in the retrieved list as $dist_k$, then the weight of each label in the list can be calculated as Equation (12), as discussed by Gou et al. [GDZX12], where $dist_i$ is the distance of bug report *i*.

$$w_i = \begin{cases} \frac{dist_k - dist_i}{\mathrm{dist}_k - dist_1}, & if \ dist_k \neq dist_1 \\ 1, & if \ dist_k = dist_1 \end{cases} \quad (12)$$

Equation (12) ensures that higher weights are given to bug reports that are closer to the incoming bug report. We need to update the score function in Equation (11) to incorporate the weights calculated in Equation (12). The updated score function is shown in Equation (13) [PF13]:

$$\text{Score}\ (c_j, N) = \textstyle\sum_{Y \in N} w(x,y) \times [class(y) = c_j] \ (13)$$

where *w(x,y)* is the weight of each instance in the top k similar returned instances which is calculated by Equation (12) according to its distance to the incoming bug report B_i corresponding to instance *X*. After calculating the weight of each label, according to Equation (13), the label with the highest weight is selected as the output label.

For using the K nearest neighbour method, we assume that the sets of bugs created by grouping bug reports based on their severity label in the dimensional space defined based on Equation (9) has the following characteristics. The distance between each bug B_i and itself is zero. The distance between B_i

and B_j is the same as the distance between B_j and B_i. If the distance between bug B_i and B_j is close and the distance between B_j and B_k is close, then the distance between B_i and B_k is also close.

6.3.6. Overall Approach

We used a simple linear model with a gradient descent optimizer in Section 6.3.1 because we want to understand the importance of similarity of features. Unlike other models such as neural networks, which tend to be hard to interpret, the trained linear model with a gradient descent optimizer could be easily interpreted to show the importance of features. Considering that our online approach, for each incoming bug report, needs to compare the incoming bug report to all previous bug reports in the dataset and we aimed to make the proposed approach easily deployable in practice, we choose to use K nearest neighbour as the classification method.

We explain the Training and Testing phase of our online severity prediction method in this Section. Figure 25 shows the overall approach.

6.3.6.1. Training

Our training phase is shown in Figure 25. The training phase starts by extracting the main features (Section 6.3.3) that can be used to predict bug severity, namely, the bug stack traces and the bug categorical features. Then, stack traces are mapped to weighed feature vectors. The details of how to create these feature vectors for stack traces are explained in Section 5.1. After creating the feature vectors for stack traces, similarity analysis (Equation 8) is used to measure the similarity between stack traces of different bug reports. We use linear combination (Section 6.3.1) of the corresponding stack traces and categorical features similarity to take categorical features into consideration in the final similarity results. The linear model is trained using the training dataset build according to Figure 24 using the 10% of the dataset and optimized using the gradient descent (Figure 16). The tuned coefficients (w_1, w_2, w_3, w_4) are then used in the testing phase.

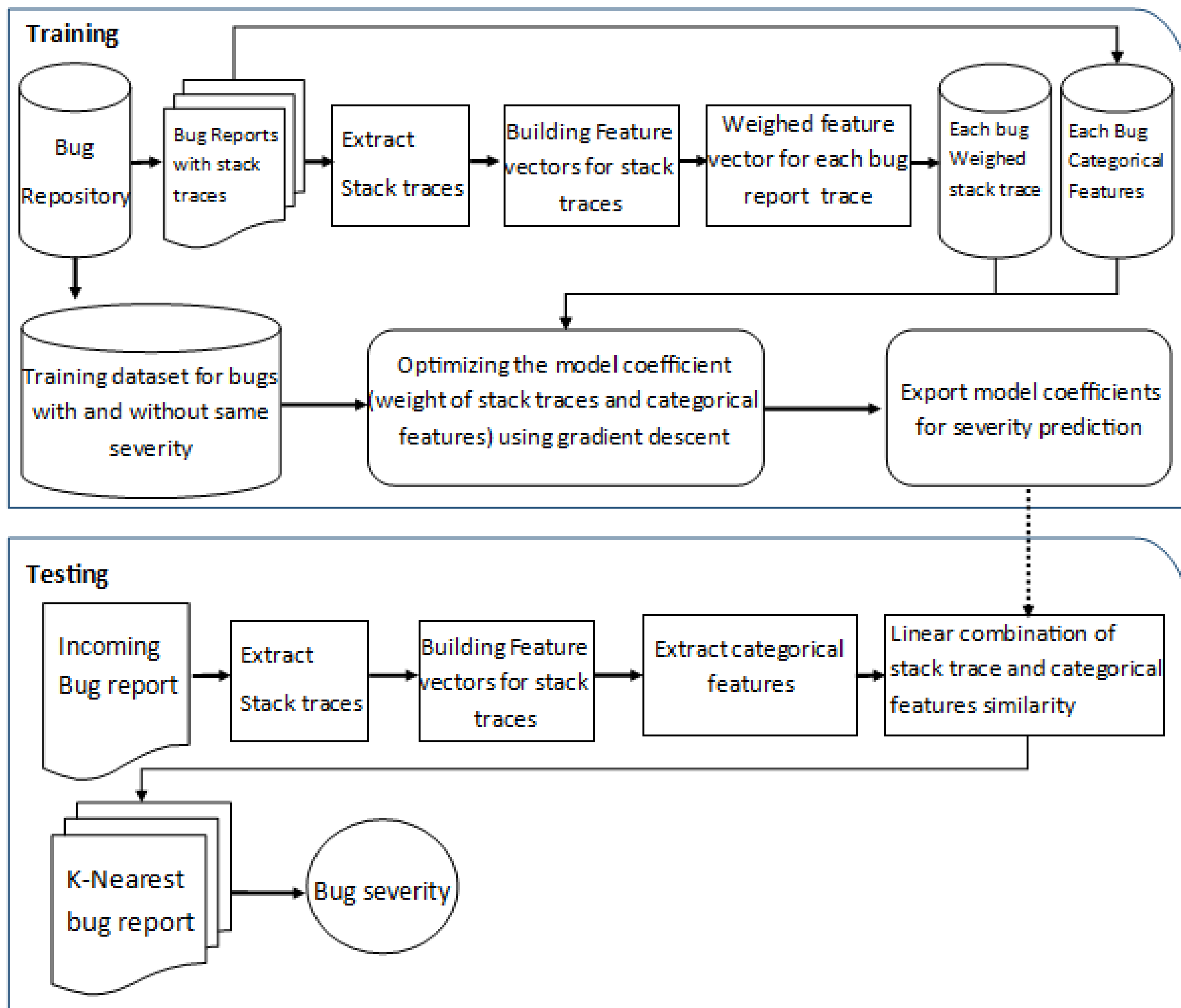

Figure 25 Overall Approach

6.3.6.2. Testing

Our online severity prediction (depicted in Figure 25) works as follows: when the system receives a new bug report, the bug stack trace is extracted using the same regular expression used in the training part. Next, the feature vector is built using all the distinct functions in the stack trace of the new bug report and stack traces of all bug reports previous to that in the bug tracking system. The feature vector is weighed using TF-IDF. After that, the similarity between the weighed feature vector of the new bug report and the weighed feature vectors of the previous bug reports in the dataset (i.e., adjacency matrix representing all stack traces of previous bug reports) is calculated. The output of this part is a matrix, which shows how similar is the incoming bug-report stack trace to the stack traces of all the previous bug reports. The

similarity of the incoming bug report stack trace is then linearly combined with the similarity of its categorical features to all the previous bug reports using the tuned coefficients $(\mathrm{w}_1, \mathrm{w}_2, \mathrm{w}_3, \mathrm{w}_4)$ from the training phase. Finally, the linear combined similarity is used to predict the severity of the incoming bug using the K-nearest neighbour method.

6.3.6.3. An example of the approach

In this Section we give an example of the approach. Let's consider we have a set of 1000 bug reports sorted based on their creation date. Dataset = $\{B_1, B_2, \dots \dots, B_{1000}\}$. In the first phase, we use 10% of the dataset for training $Dataset_{bugs} = \{B_1, B_2, \dots \dots, B_{100}\}$. Let's consider we have two severity labels $S_1 = \{ B_1, B_3\}$ and $S_2 = \{ B_2, B_4\}$. Based on Figure 24 we create our training dataset containing triples in the form of (bug, rel, irr), $Dataset_{training}$ = {(B_1, B_3, B_2), (B_1, B_3, B_4), (B_2, B_4, B_1), (B_2, B_4, B_3)}. Our training dataset with 100 bug reports has more bugs with more severity labels, but for simplicity, we gave an example of two labels and four bug reports. Next, our training model will be optimized using gradient descent (Figure 16). Let's consider the output of the training phase is that the weight of stack traces similarity importance to be w_1, the weight of products similarity importance to be w_2, the weight of components similarity importance to be w_3 and the weight of operating systems similarity importance to be w_4. Next, we use the weights to predict the severity of bugs.

To predict the severity of bugs, we used bug reports which were not included in tuning the similarity weight $(\mathrm{w}_1, \mathrm{w}_2, \mathrm{w}_3, \mathrm{w}_4)$. In our example $Dataset = \{B_{101}, B_{102}, \dots \dots, B_{1000}\}$. The dataset is sorted based on the creation date of bug reports. In the online approach with the incoming of each bug reports, it's TF-IDF similarity (based on the Equation (9)) to all the previous bug reports in the dataset is calculated.

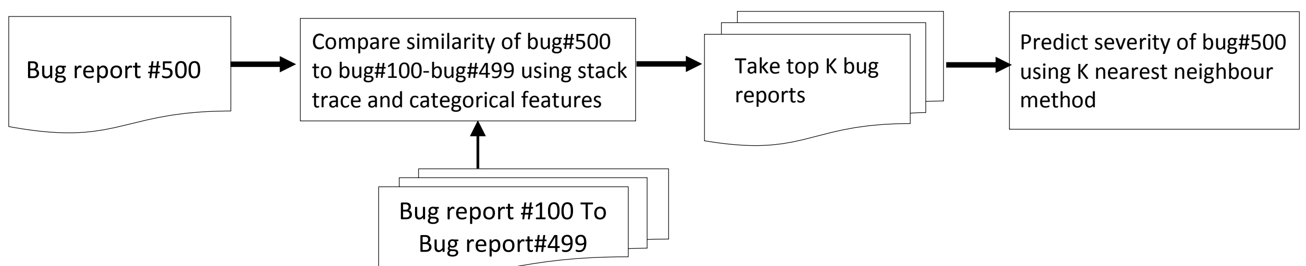

Figure 26 Example of online severity prediction approach

In this example let's consider B_{500} is reported to the bug tracking system. Based on Figure 26, the similarity of B_{500} to B_{100} - B_{499} is calculated. The similarity is defined as TF-IDF similarity of stack traces and similarity of product, component and operating system of B_{500} to B_{100} - B_{499}. Based on the calculated similarities, K nearest bug report is chosen. Next, based on the K nearest neighbour the severity of the B_{500} is predicted.

6.4. Evaluation

The goal of this Section is to evaluate the accuracy of predicting the severity based on the approach which uses stack traces and based on the approach which uses stack trace and categorical features. Both proposed approaches are then compared to the approach which uses bug report descriptions. More precisely, experiments aim to answer the following questions:

RQ1. How much improvement (if any) could be obtained by using stack traces over the approach that uses bug report descriptions?

RQ2. How much improvement (if any) could be obtained by using stack traces over the random classifier approach?

RQ3. Using the KNN classifier for predicting bug severity, how the different number of neighbours can affect the F-measure of the approach which uses stack traces?

RQ4. How adding categorical features to the stack traces affects the accuracy of the severity prediction compared to solely using stack traces and compared to using bug report descriptions?

RQ5. How adding categorical features to the stack traces affects the accuracy of the severity prediction compared to the random classifier approach?

RQ6. Using the KNN classifier for predicting bug severity, how the different number of neighbours can affect the F-measure of the approach which uses stack traces and categorical features?

Answering question 1,2 shows the importance of using stack traces as an alternative to using bug report descriptions and random approach. Answering question 3 shows the sensitivity of the proposed approach based on the number of nearest neighbours chosen. Answering question 4,5 explains how adding categorical features can contribute to predicting the severity of bugs. Answering question 6 shows the sensitivity of the approach which uses stack traces and categorical features to the number of nearest neighbours.

6.4.1. Experimental Setup

In our experiments, we compared our approaches to the approach that uses the description of bug reports. In this Section, we describe the dataset used in the experiment and provide some statistical analysis regarding the dataset.

6.4.1.1. The Datasets

The datasets used in this chapter consists of bug reports of the Eclipse and Gnome bug repositories. Both of these datasets are used extensively in different studies [YZL14, BINF12, ZYLC15, SHH16].

In these repositories, stack traces are embedded in the bug report descriptions. As explained earlier, to extract the content of the stack traces in Eclipse, we use the same regular expression presented by Lerch et al. [LM13], and for Gnome, we use the regular expression showed in Figure 6.

Normal severity is the default choice when submitting a bug report, so it is usually chosen arbitrarily. Consistent with previous severity prediction studies (e.g. [LDSV11, AADPKG08, MM08]), we remove bugs with normal severity. Furthermore, Enhancements are not considered as bugs and are removed from datasets when performing the experiments [LDSV11].

Triagers analyze bug reports in the bug tracking system to assess the validity of bug report fields. Since users are not expert in the software system, triagers further adjust and reassign bug reports fields based on the bug report description provided by the user. Consistent with the previous studies [MM08,

LDGG10, LDSV11, YHKC12, TLS12, YZL14, ZYLC15], we use the severity of bugs from the bug tracking system as the label. These severities are set by users and then further adjusted by bug triagers after the user reports the bug. Figure 27 shows the process of assessing bug severities by the triagers after the bug report is submitted by the user [SSG15].

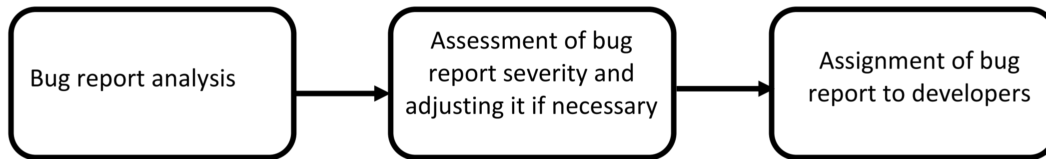

Figure 27 The bug handling process

We used the Eclipse dataset for the period of October 2001 to February 2015, having a total of 455,700 bug reports, of which 297,151 had Normal severity and 66,873 were labelled with Enhancement severity. Also, 1,000 bug reports were removed from the dataset history by the Eclipse Bugzilla administrators for maintenance purposes. After removing bug reports with Normal and Enhancement severity, we have 90,676 bug reports with severities other than Normal or Enhancement. After applying regular expression, we had 11,925 bug reports having at least one stack trace in the description. Stack traces with less than three functions are removed since they are partial stack traces and may mislead the approach. The resulting dataset contains a total of 11,825 bug reports having 17,695 stack traces in their descriptions.

The Gnome dataset used in this chapter contains bug reports from February 1997 to August 2015. The dataset contains 752,300 reports, out of which 57,446 are labelled as enhancement and 297,831 are labelled as normal. Also, 54,500 bug reports were later removed from the dataset (see Table 9). From the remaining 342,523 bug reports, 153,385 bug reports come with one or more stack traces in their description. After eliminating bug reports with partial stack traces, we had 153,343 bug reports with at least one stack trace in their description.

Since in this chapter, we removed bug reports with normal and enhancement severity, the number of bug reports with stack traces is different from the number of bug reports with stack traces presented in Table 1 and Table 2 for both datasets.

Table 9 Characteristics of the datasets

Data	Eclipse	Gnome
Total number of reports	455,700	752,300
Total number of enhancement reports	66,873	57,446
Total number of bug reports with normal severity	297,151	297,831
Total number of bug reports removed from dataset	1000	54,500
Total number of bug reports excluding normal and Enhancement	90,676	342,523
Total number of bug reports with stack traces	**11,925**	**153,343**

6.4.1.2. Dataset Setup

In our approach, we sort bug reports by their creation date and use the first 10% of the datasets to train the linear combination model of Equation (9) and optimize the coefficients using Figure 16. We tested different training dataset sizes for both datasets and did not find noticeable differences for coefficients of Equation (9) by increasing training dataset size beyond 10% of the bug reports. After optimizing the coefficient, we used the remaining 90% of the dataset for testing our online approach, as explained in Section 6.3.6.2. According to our method, bug reports are sorted based on the creation date in the bug tracking system, with the arrival of each bug report in the test set its stack trace is compared to stack traces of all previous bug reports in the test set and then using the obtained coefficients in the training phase, the similarity of the bug reports is calculated, and the k-nearest neighbour approach is then used for predicting bug severity. For instance, assume our test set has N bug reports, in our online severity prediction method with the arrival of M'th bug report ($0<M<=N$), all the M-1 previous bug reports in the test set are compared to that bug report and their similarity is calculated using the linear model of Equation (9). The K nearest neighbour is then used to predict the severity of the M'th incoming bug.

Since bug reports follow a temporal order based on their creation date, we did not use the traditional 10-fold cross-validation to validate our model. To be practical in the sense that it can be deployed in an actual bug tracking system, we create an ordered list of bug reports based on their creation date in the

bug tracking system. When predicting the severity of an incoming bug report, we compare it, using our linear model, to bug reports that were reported previous to the incoming bug report only.

6.4.1.3. Dataset Analysis

The distribution of severities of bugs in both datasets is not balanced. Overall, more bugs are having Critical or Major severities than other severity labels. The distribution of the severity labels in our datasets is shown in Figure 28. Note that for Gnome, the severity label distribution is shown using the logarithmic scale since the dataset is much more unbalanced compared to Eclipse.

These figures show that the distribution of severity labels in Eclipse and Gnome are unbalanced, favouring Critical and Major Severity labels. In the next Section we discuss the approach that is used to tackle the unbalance dataset distribution problem.

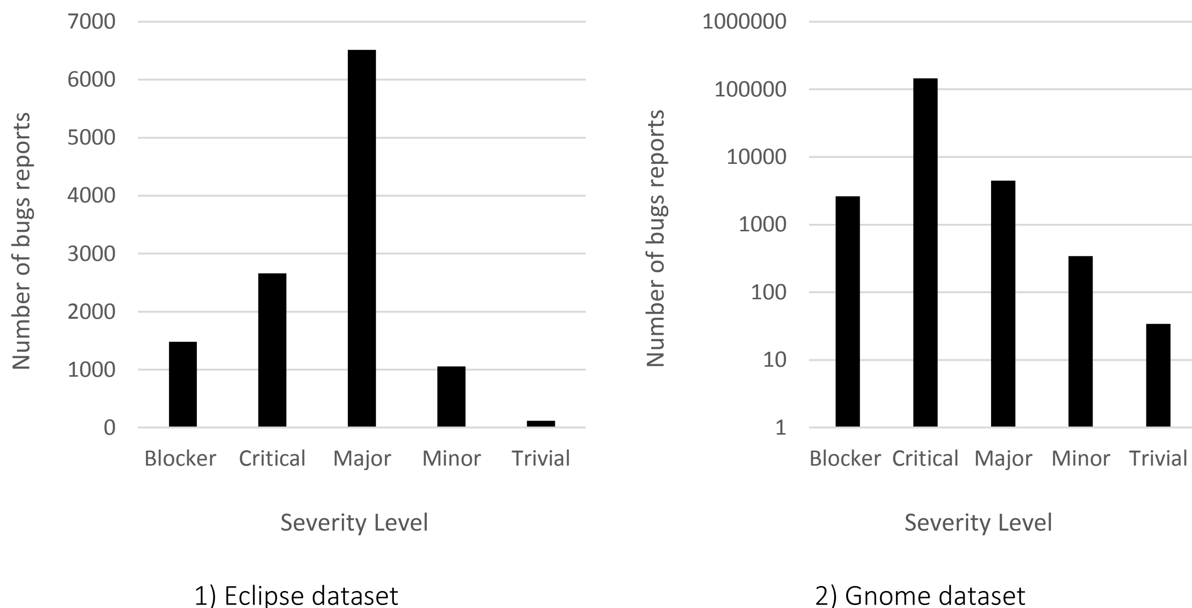

Figure 28 Distribution of the severity labels in Eclipse and Gnome datasets

6.4.2. Cost-sensitive K Nearest Neighbour

In an ideal scenario, the distribution of labels in the training set should be balanced (there are similar sample sizes for each label). Unfortunately, this scenario is not common for large industrial systems. For

example, in Eclipse and Gnome datasets, some severities have fewer bug reports in the bug tracking system and the distribution of the labels is unbalanced.

Training a classifier on an unbalanced dataset makes it biased toward the majority class labels. This is due to the fact that the classifier tends to increase the overall accuracy, which leads to ignoring minority class samples in the training set. Different approaches exist to overcome the unbalance dataset problem. These approaches include oversampling the minority class, under-sampling the majority class or creating a cost-sensitive classifier [ZM03]. We experimented with all these approaches and observed that cost-sensitive classification [ZM03] is the most suitable approach to overcome the unbalanced dataset problem in Eclipse and Gnome datasets.

To transform a classifier into a cost-sensitive classifier, we need the output of the classifier to be equal to the probability of a bug belonging to each severity class. Furthermore, we need to define a cost matrix. Using a cost matrix, the probability of each label is replaced by the average cost of choosing that class label. Indeed, to change a classifier to a cost-sensitive classifier [ZM03], we don't need to change the internal functionality of the classifier. Instead, according to the output probabilities and using a cost matrix, the classifier makes an optimal cost-sensitive prediction [ZM03].

In Equation (13), the K-nearest neighbour returns weight for each label, then the label with the largest weight is chosen. To make our classifier cost-sensitive, in the first step, we need to adjust the outputs to represent probabilities instead of weights. For example, if *B* is a bug report in a testing dataset that has *m* classes. The classifier must provide a list of probabilities $p_1 \ldots \ldots \ldots p_m$, in which each p_i shows the probability that the bug (*B*) severity label belongs to the ith class. Since the summation of all probabilities should be equal to one (i.e., $p_1 + p_2 + \cdots + p_m$ =1), the weights need to be normalized.

To calculate the probability of each label, considering that the output of our classifier is $w_1 \ldots \ldots \ldots w_i$ we use Equation (14) [SHH16], where W=$w_1 + \cdots + w_m$.

$$p_i = \frac{w_i}{W} \qquad (14)$$

The probability of each label is then changed to the classification cost of each class label by a cost matrix which contains the misclassification cost of each class label. We set the misclassification cost in a cost matrix that corresponds to the confusion matrix [LD13] in Figure 29.

	Actual	
	Positive	Negative
Predicted	True positive	False positive
	False negative	True negative

Figure 29 Confusion matrix

In the confusion matrix, higher values of true positives and true negatives are favourable, thus we set the misclassification cost of these two values to zero. However, the cost of false positives and false negatives is selected based on the classification context.

We overcome the unbalance distribution problem by assigning high misclassification costs to the under-sampled class labels and low misclassification cost to over-sampled class labels. We chose the cost of the misclassification of each class label to be reciprocal to the number of existing instances of that class divided by the number of instances of the majority class (see Equation (15)). In this case, classes which have a lower number of instances will have higher misclassification cost and classes which have a high number of instances will have lower misclassification cost. If we consider having *C* different classes and assume s_j is the number of instances of class *j* in the training set and *s* is the number of instances of the majority class in the training set, then the misclassification cost of each class C_j is calculated by Equation (15).

$$MC_j = \frac{s}{s_j} \qquad (15)$$

The cost matrix is constructed using the misclassifications cost based on Equation (15). Then, we need to calculate the classification cost of each class label based on the cost matrix and the calculated probabilities based on Equation (14). Assume we have *M* classes and the incoming bug report belongs to each of these classes with probabilities of $P_1 \dots \dots P_m$, and assume that each class has a misclassification cost of $CO_1 \dots \dots CO_m$, then the cost of assigning the bug report to each of those classes is calculated by (16) [QWZZ13].

$$CCO_i = \sum_{j \in m \; and \; j \neq i} CO_j \times P_j \quad (16)$$

Finally, the class label with the lowest classification cost is selected as the output of the classifier. Misclassification costs, if assigned improperly, may degrade the classification accuracy of the classifier. In this chapter, we used Equation (15) to determine the misclassification costs. Furthermore, to avoid very high misclassification costs that may be associated with some class labels, we choose a threshold of ten to be the maximum misclassification cost.

For example, let us assume the misclassification cost of Blocker, Critical, Major, Minor and Trivial are 4.3, 2.5, 1, 6.2 and 10 respectively. Let us also assume that after applying K-nearest neighbour and calculating probabilities based on Equation (14), the bug severity is 20% Blocker, 10% Critical, 30% Major, 30% Minor, and 10% trivial. According to Equation (16), we will have the cost of predicting each severity label and choose the label with the lowest classification cost as follows:

$$Cost_{Blocker} = 0 * 0.2 + 2.5 * 0.1 + 1 * 0.3 + 6.25 * 0.3 + 10 * 0.1 = 3.425$$
$$Cost_{Critical} = 4.3 * 0.2 + 0 * 0.1 + 1 * 0.3 + 6.25 * 0.3 + 10 * 0.1 = 4.035$$
$$Cost_{Major} = 4.3 * 0.2 + 2.5 * 0.1 + 0 * 0.3 + 6.25 * 0.3 + 10 * 0.1 = 3.985$$
$$Cost_{Minor} = 4.3 * 0.2 + 2.5 * 0.1 + 1 * 0.3 + 0 * 0.3 + 10 * 0.1 = 2.41$$
$$Cost_{trivial} = 0 * 0.2 + 2.5 * 0.1 + 1 * 0.3 + 6.25 * 0.3 + 0 * 0.1 = 2.985$$

Figure 30 cost of predicting each severity label

Based on the proposed cost-sensitive K nearest neighbour method, we update our testing phase to take into account the probability of each severity and the cost of classifying each bug to that severity using the misclassification cost matrix. Finally, the severity with the least classification cost will be selected as the output label (Figure 31 shows this process).

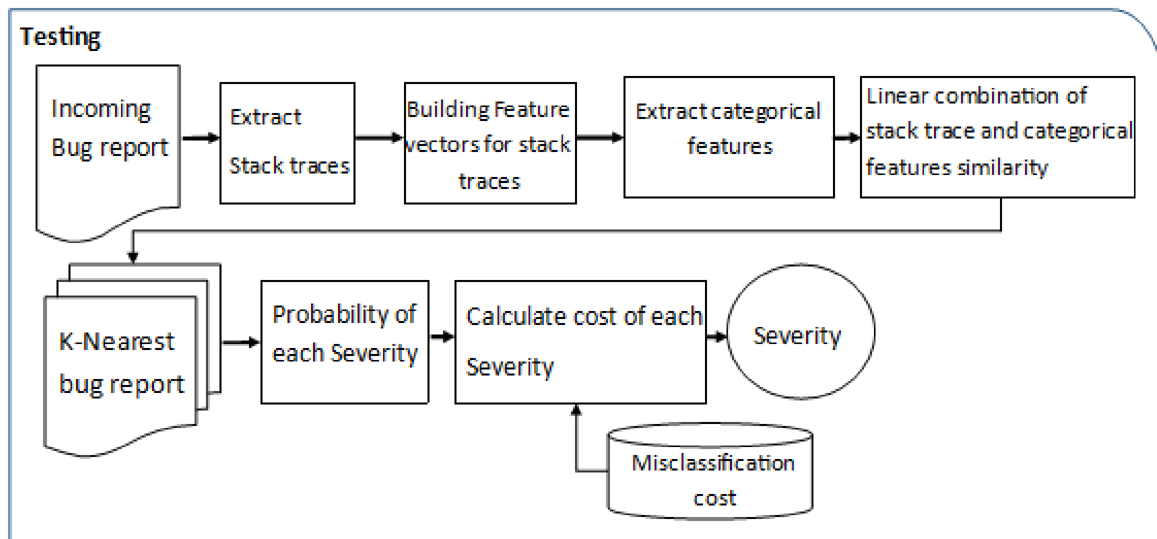

Figure 31 Updated testing phase using cost-sensitive k nearest neighbour

6.4.3. Predicting Severity of Bugs Using Description

To predict the severity of a bug using the bug report description, we extract descriptions from all bug reports in our dataset. We tokenize words in the description of bug reports by splitting the text using space and new line character. We used the raw tokenized words for building a feature vector for each bug report. We build the feature vector using the distinct words in all bug report descriptions in our dataset.

We use TF-IDF to weight the feature vectors, just as we did in our approach (Section 6.1). In this approach, each bug report description is represented by a vector built based on the frequency of occurrence of each word (Equation (2)) multiplied by inverse document frequency of each word (Equation (3)). We follow the same online severity prediction approach, discussed in (Section 6.3.6.2) for predicting severity based on weighed feature vectors constructed from bug report descriptions.

We create an ordered set of bug reports based on their creation date. We then exercise the scenario in which each bug report in our ordered set is compared to all previous bug reports using their corresponding weighed feature vectors. After calculating the distance of each bug report to all the previous bug reports, we use the K nearest neighbour bug reports to determine the bug severity label using (Equation (13)). Based on the K nearest neighbour of (Equation (13)), the severity of an incoming bug report is selected based on the severity label of its nearest neighbours weighed using their distance to the incoming bug report.

Furthermore, we tackle the unbalanced dataset distribution problem by using the cost-sensitive k nearest neighbour method with the same setting as for our approach using stack traces and categorical features. In this approach after calculating the probability of each severity label, we use cost-sensitive k nearest neighbour of Equation (16) which calculates the cost of choosing each severity label by considering the misclassification cost of each severity label calculated using Equation (15) to predict the severity of a bug. For example, assume our test set contains sorted bug reports $\{B_M, \dots \dots, B_N\}$, to predict the severity of B_J (M < J < N), we compared B_J description to all the bug reports in $\{B_M, \dots, B_{J-1}\}$, then we predict the severity of B_J using the Equation (16).

6.4.4. Predicting Severity of Bugs Using a Random Classifier

We also compare the result of the approach, which uses stack traces and the approach which uses stack traces and categorical features to a random classifier, which selects a severity label for each bug in proportion to the different class labels. Assume we have N severity classes and the number of bug reports that belong to each class is $B_{S_1} \dots \dots B_{S_N}$. If we randomly select severity labels for each bug, then our accuracy of predicting the severity label S_i can be calculated using (17):

$$Accuracy\ (S_i) = \frac{B_{S_i}}{\sum_{j=1}^{N} B_{S_j}} \quad (17)$$

6.4.5. Severity Prediction Approaches Setup

For all the models, we followed the same approach to address the unbalanced dataset distribution problem. We explained the heuristic used for choosing misclassification costs in Section 6.4.2. For all the models, the returned list size of the k-nearest neighbour varies from 1 to 10 to assess the severity prediction accuracy with different returned list sizes. For the approach that use stack traces and categorical features, we initialized the four weights (coefficients initial value) from a normal distribution with zero mean and a standard deviation of 0.1 and used 0.001 as the learning rate. We also trained the model using the first 10% of the dataset as explained in Section 6.6.2.

6.4.6. Evaluation Metrics

In this study, we use the precision, recall and F-measure metrics to evaluate our approach. If we consider the number of bugs for which we predict that they should have a severity label S_L as P_{S_L} and the number of bugs for which we correctly predict the severity label to be S_L as C_{S_L} then the precision is defined by Equation (18):

$$Precision\ (S_L) = \frac{C_{S_l}}{P_{S_l}} \quad (18)$$

Furthermore, if we consider the number of bugs that actually have the severity label S_L (ground truth) as T_{S_L} then recall is calculated by Equation (19):

$$Recall\ (S_L) = \frac{C_{S_l}}{T_{S_l}} \quad (19)$$

In this study, a confusion matrix is built for each severity label. Then using the built confusion matrix, recall and precision are calculated. Since precision is the ratio of correctly predicted labels of a specific severity to the total number of labels predicted to have that severity, it actually measures the correctness of the approach. Meanwhile, since recall is the number of correct predictions of a severity to the total number of instances of that severity, it actually shows the completeness of the approach. However, the common practice is to combine these two metrics together to have a better perception of the accuracy

of the severity prediction results. A common approach for combining these two metrics is F-measure. F-measure is the harmonic mean of precision and recall [GS14]. F-measure is calculated according to Equation (20).

$$F - measure = \frac{2 \times Precision \times Recall}{Precision + Recall} \qquad (20)$$

6.4.7. Evaluation Results

In the rest of the chapter, we refer to the approach that uses stack traces alone as BSP_{ST}, the approach that uses bug report descriptions as BSP_{DE}, and the approach that uses stack traces and categorical features as BSP_{ST+CF}. The approach that uses bug report descriptions predicts the severity of bug reports using the description of bug reports, which also include stack traces since stack traces are copied inside bug report descriptions.

In this section, we discuss the results of applying the proposed approach to stack traces and categorical features of the bug reports of the Eclipse and Gnome datasets. We show that BSP_{ST} outperforms BSP_{DE}. We also show how BSP_{ST+CF} performs better compared to BSP_{ST} and BSP_{DE}. When using a K-nearest neighbour classifier, one of the most important factors is the value of K. The value of K shows the number of most similar items, which are used to choose the label. In our experiments, we recorded precision and recall and calculated F-measure by varying the value of K from 1 to 10 for each severity label in each dataset.

Figure 32 to Figure 41 show the results for predicting different severity levels by varying list size for the Eclipse and Gnome datasets, respectively. We revisit the research questions in light of the results presented in Figure 32 to Figure 41. We provide the detailed values of precision, recall and F-measure for each of the list sizes for all severity levels for both datasets in Table 10 to Table 19.

6.4.7.1. F-measure of BSP_{ST}

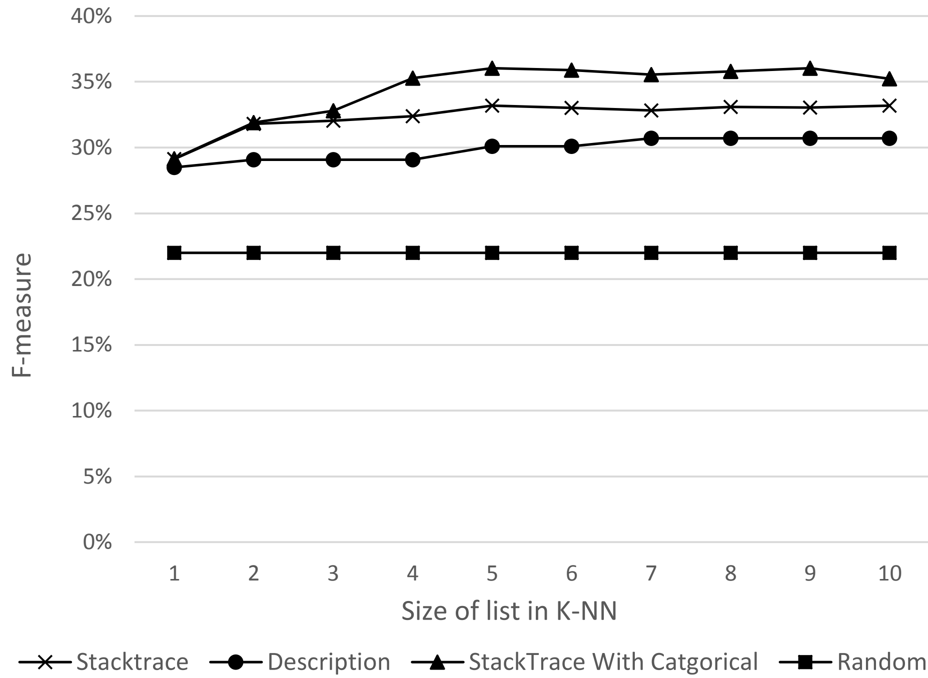

Figure 32 F-measure of predicting different severity levels by varying list size in Eclipse dataset – Critical Severity Level

Table 10 Severity prediction accuracy (Eclipse Critical severity)

List size	Bug report description			Stack traces			Stack trace and categorical features		
	Precision	**Recall**	**F-measure**	**Precision**	**Recall**	**F-measure**	**Precision**	**Recall**	**F-measure**
1	28.2%	29.1%	28.64%	31.5%	27.1%	29.1%	34.5%	25.1%	29.14%
2	26.4%	33.1%	29.37%	30.6%	33.1%	31.8%	33.9%	30.1%	31.88%
3	26.3%	33.1%	29.31%	30%	34.4%	32%	34.3%	31.4%	32.785
4	26.3%	33.1%	29.31%	30.3%	34.8%	32.4%	33.3%	37.5%	35.27%
5	27.5%	34%	30.4%	31.1%	35.6%	33.2%	34.1%	38.2%	36.03%
6	27.2%	34.1%	30.26%	31.1%	35.2%	33%	33.3%	38.9%	35.88%
7	28%	34%	30.7%	31.3%	34.5%	32.8%	33.3%	38.1%	35.53%
8	28%	34%	30.7%	31.6%	34.7%	33.1%	33.3%	38.7%	35.79%
9	28%	34%	30.7%	31.6%	34.6%	33%	34%	38.3%	36.02%
10	28%	34%	30.7%	31.7%	34.8%	33.2%	32.6%	38.3%	35.22%

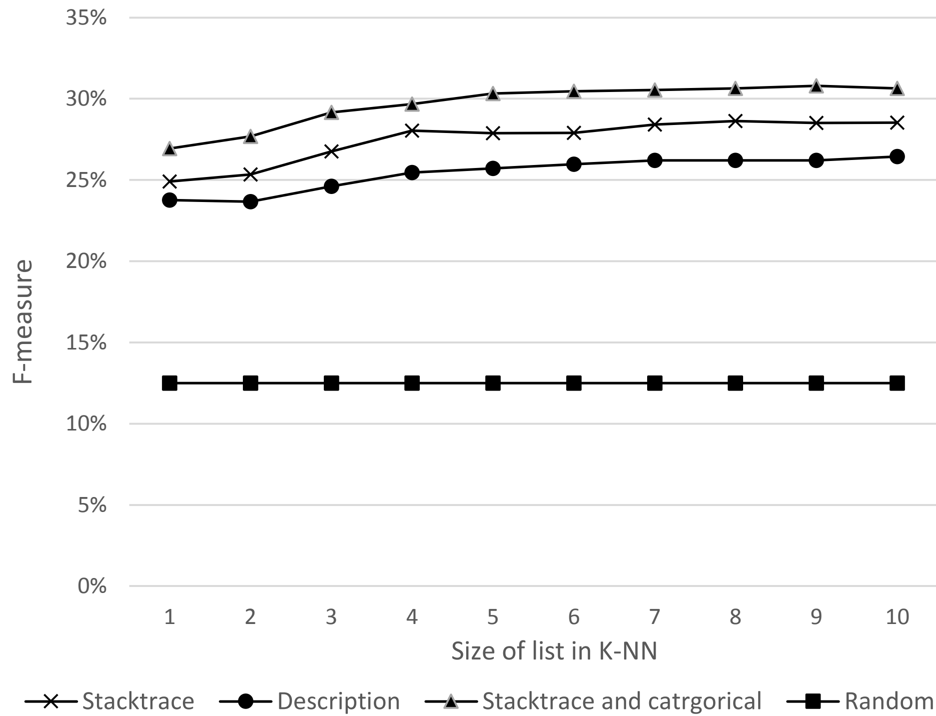

Figure 33 F-measure of predicting different severity levels by varying list size in Eclipse dataset – Blocker Severity Level

Table 11 Severity prediction accuracy (Eclipse Blocker severity)

List size	Bug report description			Stack trace			Stack trace and categorical features		
	Precision	**Recall**	**F-measure**	**Precision**	**Recall**	**F-measure**	**Precision**	**Recall**	**F-measure**
1	23%	24.6%	23.77%	26.5%	23.5%	24.91%	27.7%	26.2%	26.92%
2	20%	29.4%	23.8%	23.2%	27.9%	25.33%	24.5%	31.8%	27.67%
3	20%	32%	26.76%	23%	32%	26.76%	24.2%	36.7%	29.16%
4	20%	35.3%	28%	23.2%	35.4%	28.03%	23.8%	39.4%	29.67%
5	20%	36.1%	27.87%	22.7%	36.1%	27.87%	24.2%	40.6%	30.32%
6	20%	37.3%	27.89%	22.5%	36.7%	27.89%	24.1%	41.4%	30.46%
7	20%	38%	28.42%	22.7%	38%	28.42%	24%	42%	30.54%
8	20%	38%	28.63%	22.8%	38.5%	28.63%	23.8%	43%	30.64%
9	20%	38.1%	28.5%	22.6%	38.6%	28.50%	23.8%	43.6%	30.79%
10	20%	39%	28.53%	22.5%	39%	28.53%	23.5%	44%	30.63%

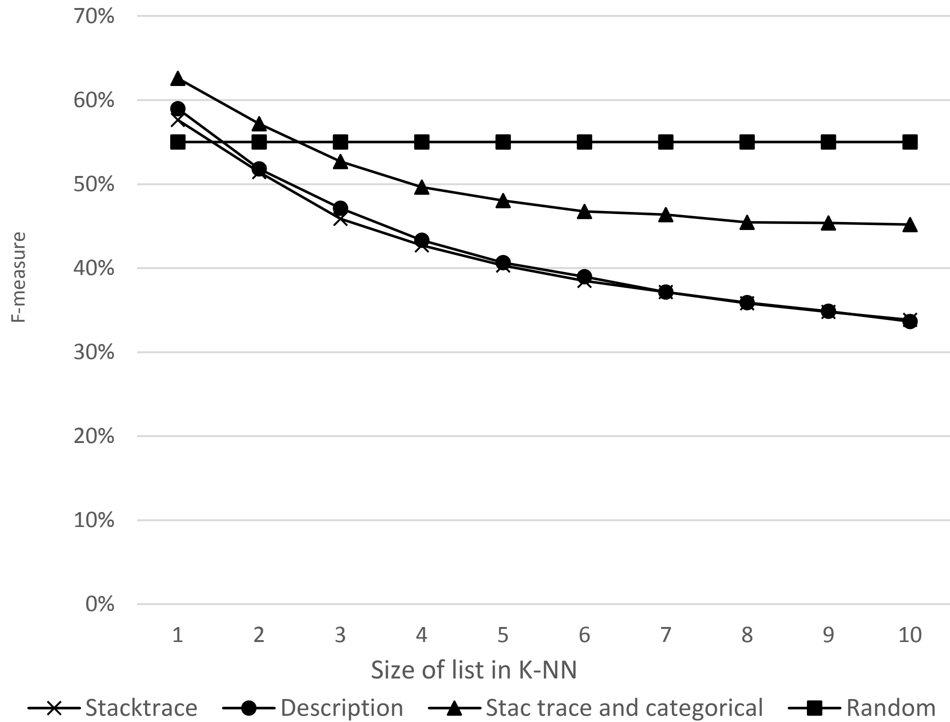

Figure 34 F-measure of predicting different severity levels by varying list size in Eclipse dataset – Major Severity Level

Table 12 Severity prediction accuracy (Eclipse Major severity)

List size	Bug report description			Stack trace			Stack trace and categorical features		
	Precision	**Recall**	**F-measure**	**Precision**	**Recall**	**F-measure**	**Precision**	**Recall**	**F-measure**
1	60%	58%	62.59%	61.7%	54.1%	57.65%	63.1%	62.1%	62.59%
2	61.1%	45%	57.16%	63.5%	43.2%	51.41%	64.7%	51.2%	57.16%
3	62%	38%	52.69%	64.5%	35.6%	45.87%	65.9%	43.9%	52.69%
4	63.3%	33%	49.63%	65.5%	31.7%	42.72%	66.5%	39.6%	49.63%
5	63%	30.1%	48.03%	66%	29%	40.29%	66.8%	37.5%	48.03%
6	64.1%	28%	46.76%	66.3%	27.1%	38.47%	67.4%	35.8%	46.76%
7	65%	26%	46.35%	66.2%	25.8%	37.12%	67.5%	35.3%	46.35%
8	65%	24.8%	45.44%	66.5%	24.5%	35.80%	67.7%	34.2%	45.44%
9	65%	23.8%	45.37%	66.9%	23.5%	34.78%	68.6%	33.9%	45.37%
10	65%	22.7%	45.19%	66.5%	22.7%	33.84%	68.2%	33.8%	45.19%

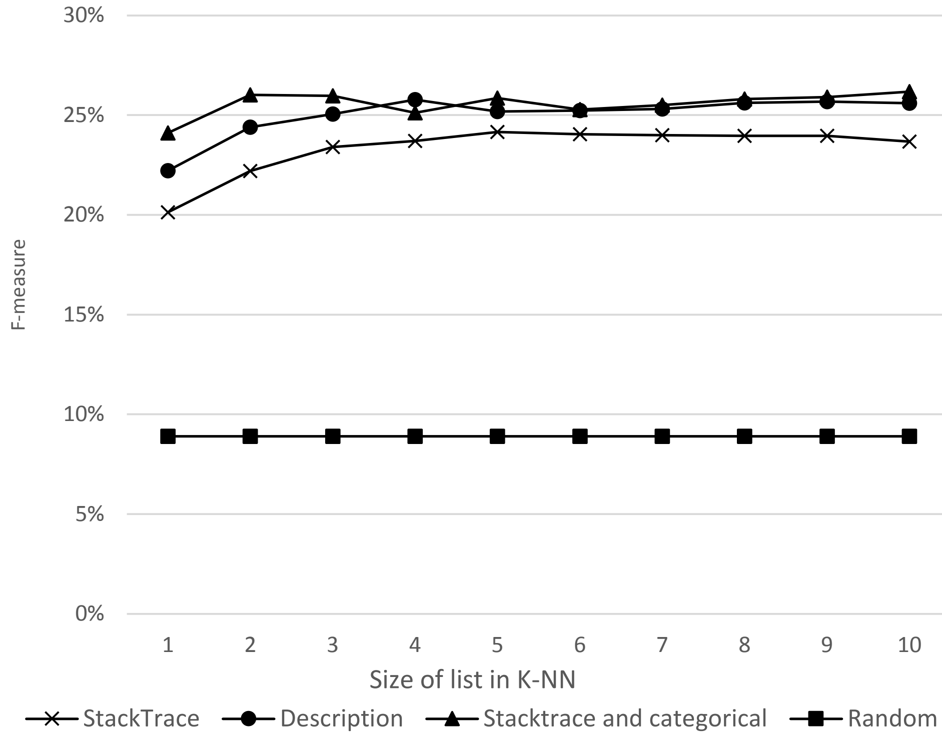

Figure 35 F-measure of predicting different severity levels by varying list size in Eclipse dataset – Minor Severity Level

Table 13 Severity prediction accuracy (Eclipse Minor severity)

List size	Bug Report Description			Stack trace			Stack trace and categorical Features		
	Precision	**Recall**	**F-measure**	**Precision**	**Recall**	**F-measure**	**Precision**	**Recall**	**F-measure**
1	23.1%	21.4%	22.21%	23.5%	17.6%	20.12%	28.1%	21.1%	24.10%
2	20.9%	29.3%	24.39%	20.8%	23.8%	22.19%	24.6%	27.6%	26.01%
3	19.6%	34.7%	25.05%	19.8%	28.6%	23.4%	22.1%	31.5%	25.97%
4	19.2%	39.2%	25.77%	18.8%	32.1%	23.71%	20.1%	33.5%	25.12%
5	18.2%	40.9%	25.19%	18.5%	34.8%	24.15%	19.9%	36.9%	25.85%
6	17.8%	43.35	25.22%	18%	36.2%	24.04%	19.1%	37.4%	25.28%
7	17.5%	45.7%	25.30%	17.6%	37.7%	23.99%	18.9%	39.2%	25.50%
8	17.6%	47.1%	25.62%	17.3%	39%	23.96%	18.7%	41.6%	25.80%
9	17.5%	48.2%	25.67%	17.1%	40%	23.95%	18.5%	43.2%	25.90%
10	17.3%	49.2%	25.59%	16.8%	40.1%	23.67%	18.6%	44.2%	26.18%

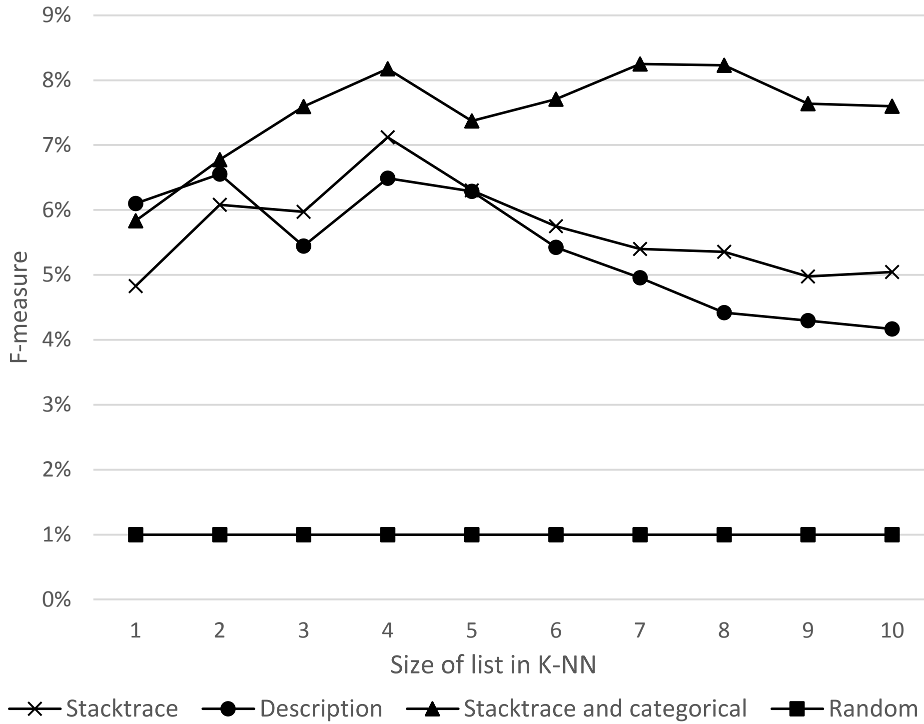

Figure 36 F-measure of predicting different severity levels by varying list size in Eclipse dataset – Trivial Severity Level

Table 14 Severity prediction accuracy (Eclipse Trivial severity)

List size	Bug report description			Stack trace			Stack trace and categorical Features		
	Precision	Recall	F-measure	Precision	Recall	F-measure	Precision	Recall	F-measure
1	6.7%	5.6%	6.10%	5.5%	4.3%	4.82%	7%	5%	5.83%
2	5.5%	8.1%	6.55%	5.5%	6.8%	6.08%	6.4%	7.2%	6.77%
3	4.1%	8.1%	5.44%	4.6%	8.5%	5.96%	6.2%	9.8%	7.59%
4	4.6%	11%	6.48%	5.1%	11.8%	7.12%	5.9%	13.3%	8.17%
5	4.4%	11%	6.28%	4.3%	11.8%	6.30%	5.1%	13.3%	7.375
6	3.6%	11%	5.42%	3.8%	11.8%	5.74%	5.2%	14.9%	7.70%
7	3.2%	11%	4.95%	3.5%	11.8%	5.39%	5.5%	16.5%	8.25%
8	2.8%	10.5%	4.42%	3.4%	12.6%	5.35%	5.4%	17.3%	8.23%
9	2.7%	10.5%	4.29%	3.1%	12.6%	4.975	4.9%	17.3%	7.63%
10	2.6%	10.5%	4.16%	3.1%	13.5%	5.04%	4.8%	18.2%	7.6%

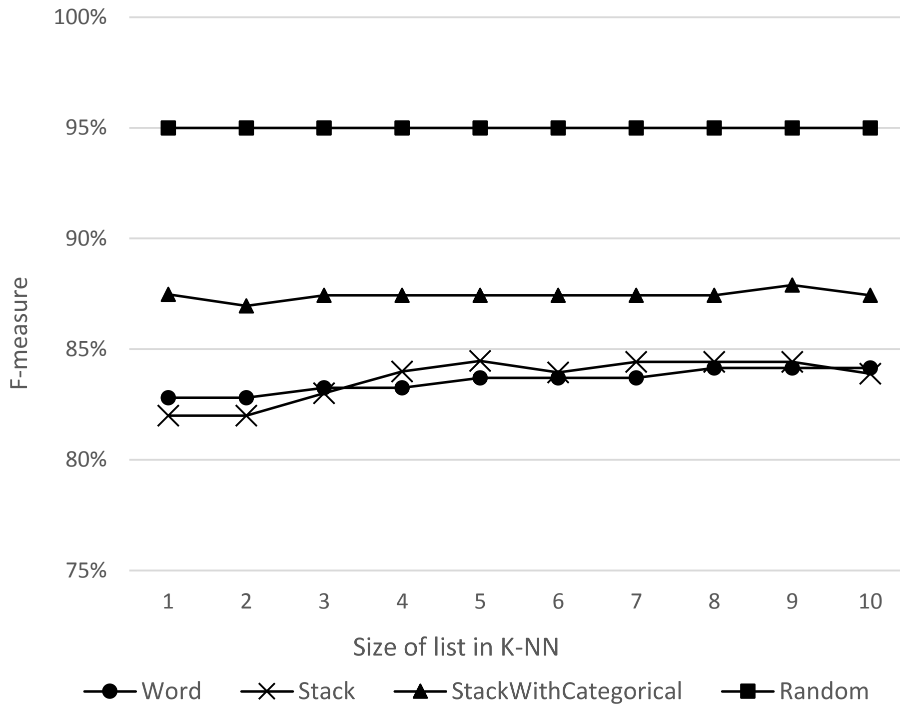

Figure 37 F-measure of predicting different severity levels by varying list size in Gnome dataset – Critical Severity Level

Table 15 Severity prediction accuracy (Gnome Critical severity)

List size	Bug report description			Stack trace			Stack trace and categorical Features		
	Precision	Recall	F-measure	Precision	Recall	F-measure	Precision	Recall	F-measure
1	79%	87%	82.80%	82%	82%	82%	86%	89%	87.47%
2	79%	87%	82.80%	82%	82%	82%	85%	89%	86.95%
3	79%	88%	83.25%	83%	83%	83%	85%	90%	87.42%
4	79%	88%	83.25%	83%	85%	83.98%	85%	90%	87.42%
5	79%	89%	83.70%	83%	86%	84.47%	85%	90%	87.42%
6	79%	89%	83.70%	82%	86%	83.95%	85%	90%	87.42%
7	79%	89%	83.70%	82%	87%	84.42%	85%	90%	87.42%
8	79%	90%	84.14%	82%	87%	84.42%	85%	90%	87.42%
9	79%	90%	84.14%	82%	87%	84.42%	85%	91%	87.89%
10	79%	90%	84.14%	81%	87%	83.89%	85%	90%	87.42%

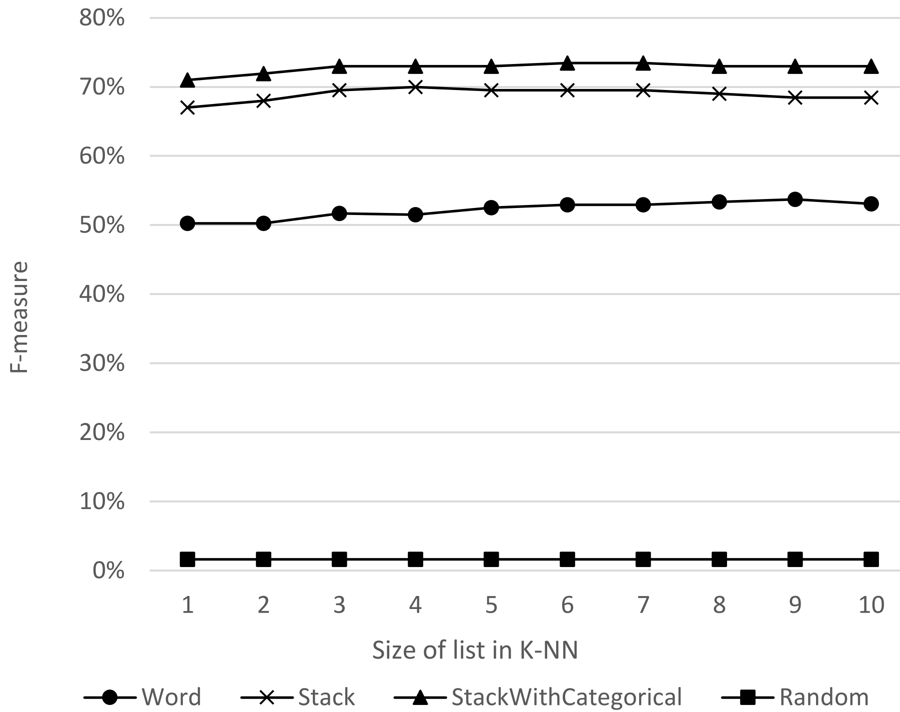

Figure 38 F-measure of predicting different severity levels by varying list size in Gnome dataset – Blocker Severity Level

Table 16 Severity prediction accuracy (Gnome Blocker severity)

List size	Bug report description			Stack trace			Stack trace and categorical Features		
	Precision	**Recall**	**F-measure**	**Precision**	**Recall**	**F-measure**	**Precision**	**Recall**	**F-measure**
1	54%	47%	50.25%	67%	67%	67%	70%	72%	70.98%
2	54%	47%	50.25%	67%	69%	67.98%	70%	74%	71.94%
3	56%	48%	51.69%	69%	70%	69.49%	72%	74%	72.98%
4	57%	47%	51.51%	70%	70%	70%	72%	74%	72.98%
5	58%	48%	52.52%	70%	69%	69.49%	72%	74%	72.98%
6	59%	48%	52.93%	70%	69%	69.49%	72%	75%	73.46%
7	59%	48%	52.93%	70%	69%	69.49%	72%	75%	73.46%
8	60%	48%	53.33%	69%	69%	69%	73%	73%	73%
9	61%	48%	53.72%	70%	67%	68.46%	73%	73%	73%
10	61%	47%	53.09%	70%	67%	68.46%	73%	73%	73%

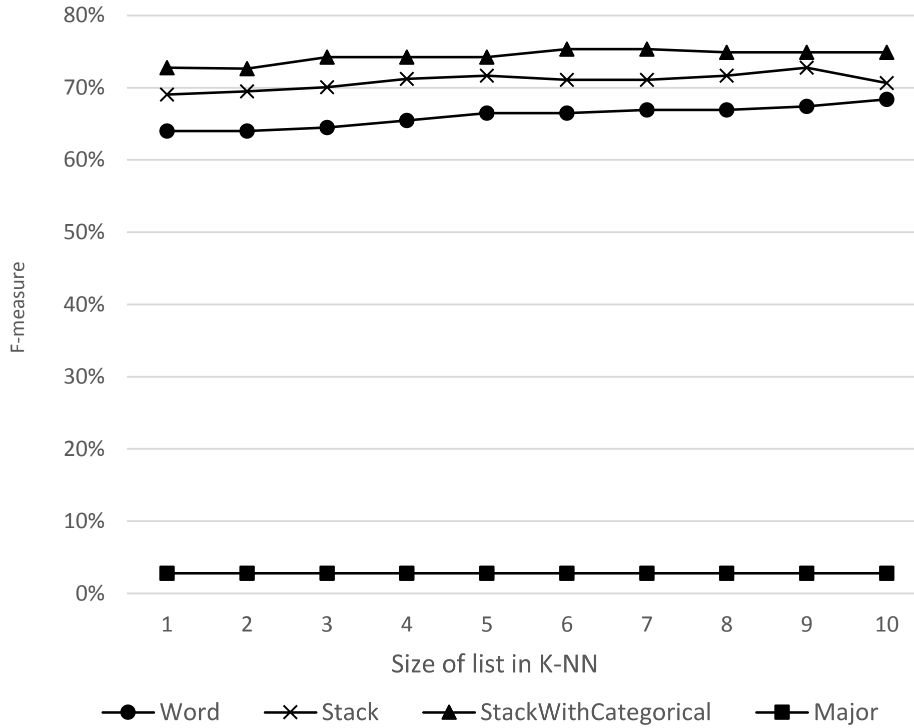

Figure 39 F-measure of predicting different severity levels by varying list size in Gnome dataset – Major Severity Level

Table 17 Severity prediction accuracy (Gnome Major severity)

List size	Bug report description			Stack trace			Stack trace and categorical Features		
	Precision	Recall	F-measure	Precision	Recall	F-measure	Precision	Recall	F-measure
1	64%	64%	64%	75%	64%	69.06%	77%	69%	72.78%
2	64%	64%	64%	76%	64%	69.48%	78%	68%	72.65%
3	64%	65%	64.49%	76%	65%	70.07%	79%	70%	74.22%
4	64%	67%	65.46%	76%	67%	71.21%	79%	70%	74.22%
5	65%	68%	66.46%	77%	67%	71.65%	79%	70%	74.22%
6	65%	68%	66.46%	77%	66%	71.07%	79%	72%	75.33%
7	65%	69%	66.94%	77%	66%	71.07%	79%	72%	75.33%
8	65%	69%	66.94%	77%	67%	71.65%	78%	72%	74.88%
9	65%	7%	67.40%	77%	69%	72.78%	78%	72%	74.88%
10	66%	71%	68.40%	76%	66%	70.64%	78%	72%	74.88%

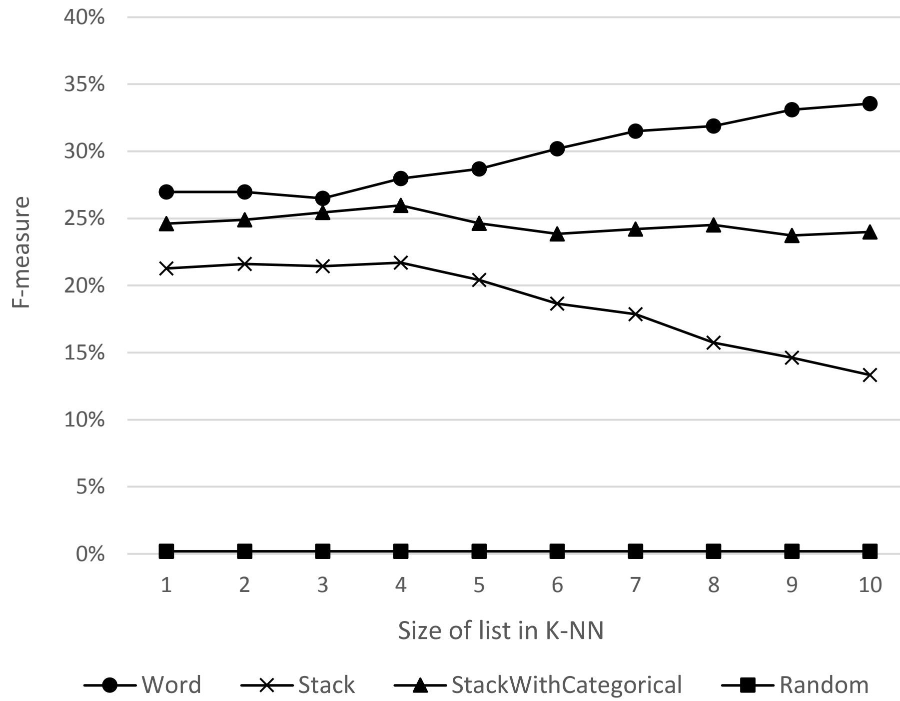

Figure 40 F-measure of predicting different severity levels by varying list size in Gnome dataset – Minor Severity Level

Table 18 Severity prediction accuracy (Gnome Minor severity)

List size	Bug report description			Stack trace			Stack trace and categorical Features		
	Precision	Recall	F-measure	Precision	Recall	F-measure	Precision	Recall	F-measure
1	26%	28%	26.96%	26%	18%	21.27%	32%	20%	24.61%
2	26%	28%	26.96%	27%	18%	21.6%	33%	20%	24.90%
3	26%	27%	26.49%	295	17%	21.43%	35%	20%	25.45%
4	29%	27%	27.96%	30%	17%	21.70%	37%	20%	25.96%
5	32%	26%	28.68%	32%	15%	20.42%	39%	18%	24.63%
6	36%	265	51.62%	33%	13%	18.65%	40%	17%	23.85%
7	40%	26%	31.51%	35%	12%	17.87%	42%	17%	24.20%
8	44%	25%	31.88%	37%	10%	15.74%	44%	17%	24.52%
9	49%	25%	33.10%	39%	9%	14.62%	46%	16%	23.745
10	51%	25%	33.55%	40%	8%	13.33%	48%	16%	24%

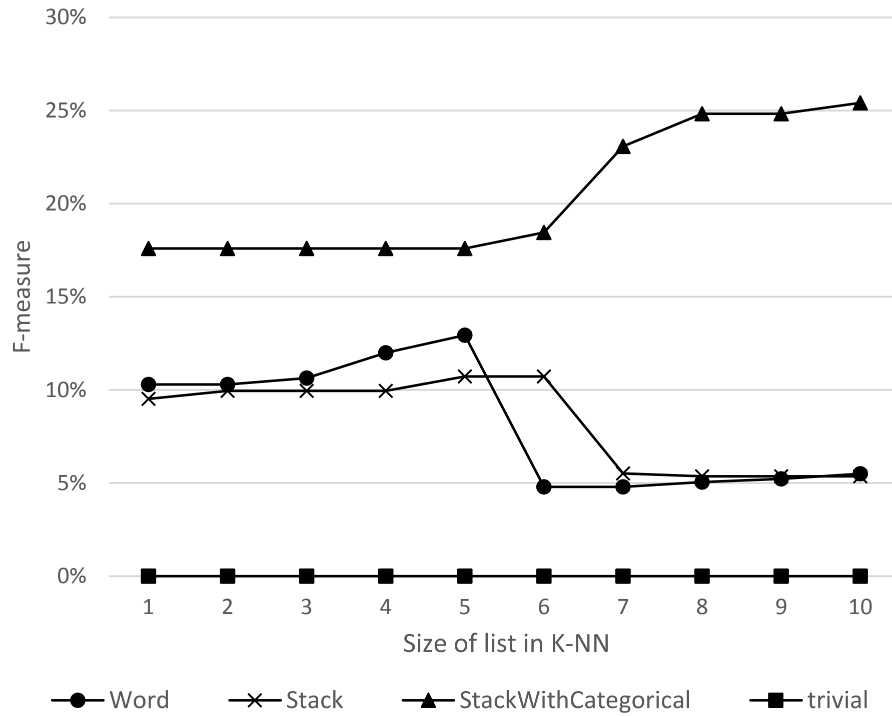

Figure 41 F-measure of predicting different severity levels by varying list size in Gnome dataset – Trivial Severity Level

Table 19 Severity prediction accuracy (Gnome Trivial severity)

List size	Bug report description			Stack trace			Stack trace and categorical Features		
	Precision	**Recall**	**F-measure**	**Precision**	**Recall**	**F-measure**	**Precision**	**Recall**	**F-measure**
1	12%	9%	10.28%	23%	6%	9.51%	33%	12%	17.6%
2	12%	9%	10.28%	29%	6%	9.94%	33%	12%	17.6%
3	13%	9%	10.63%	29%	6%	9.94%	33%	12%	17.6%
4	18%	9%	12%	29%	6%	9.94%	33%	12%	17.6%
5	23%	9%	12.935	5%	6%	10.71%	33%	12%	17.6%
6	12%	3%	4.8%	5%	6%	10.71%	40%	12%	18.46%
7	12%	3%	4.8%	34%	3%	5.51%	50%	15%	23.07%
8	16%	3%	5%	25%	3%	5.35%	72%	15%	24.82%
9	20%	3%	5%	25%	3%	5.35%	72%	15%	24.82%
10	33%	3%	5.5%	25%	3%	5.35%	83%	15%	25.40%

Figure 32 to Figure 41 show the F-measure of the BSP_{ST} compared to the F-measure of BSP_{DE}. For Eclipse dataset severity prediction using BSP_{ST} outperforms BSP_{DE} for Critical and Blocker severity labels. BSP_{ST} has the same performance as BSP_{DE} for Major and Trivial severity labels. For Gnome dataset severity prediction using BSP_{ST} outperforms BSP_{DE} for Major and Blocker severity labels. BSP_{ST} has the same performance as BSP_{DE} for Critical and Trivial severity labels.

For both datasets, BSP_{DE} outperforms BSP_{ST} for the Minor severity label only. It is worth to mention that although the bug report description contains a stack trace, we have higher accuracy using stack traces independently, as described in our approach. These results answer RQ1 and confirm that BSP_{ST} outperforms or has the same performance as BSP_{DE} for predicting all bug severity levels, except the Minor severity level, when applied to Eclipse and Gnome bug report datasets.

We further investigated the reason that BSP_{ST} is slightly less performant compared to BSP_{DE} when predicting the Minor severity level. We elaborate more on this at the end of this Section.

We also answer RQ2 by comparing the performance of our approach to a random classifier. Based on Figure 32 to Figure 41, we see that our approach outperforms a random classifier for all severity labels for both datasets, except for the Critical severity level in the case of the Gnome dataset. This is caused by two factors, including the fact that the Critical severity label is the majority class label in Gnome and that the distribution of labels in Gnome favours the majority class.

Based on Figure 28, the number of bug reports in Eclipse with Major severity is excessively higher than other severity labels. Using a classifier, the excessive number of Major severity label instances creates a bias in the outcome of the classifier by drifting the machine learning approach toward predicting a Major class label to increase the overall accuracy. This explains the similar results in the severity prediction performance using BSP_{ST} and BSP_{DE} in Figure 34.

The accuracy of predicting the Trivial severity level using both approaches is low compared to other severity labels in both datasets. The reason is due to the fact that the number of bug reports having trivial severity is considerably less than other severity labels.

6.4.7.2. Sensitivity of the Approach to the Number of Nearest Neighbours

We answer RQ3 and RQ6 Based on Figure 32 to Figure 41. Based on the results both BSP_{ST} and BSP_{ST+CF} are slightly sensitive to the number of nearest neighbours chosen. For Gnome, none of the approaches are hugely sensitive to the number of nearest neighbours. For Eclipse, only the major severity is slightly sensitive to the number of neighbours. The reason is that the Major severity label based on Figure 28 is the majority class label. In this case, increasing the number of nearest neighbours will cause more labels to appear in the returned list, which increases the probability of not choosing a majority severity label.

6.4.7.3. Severity Prediction Improvement by Adding Categorical Features

As shown in Figure 32 to Figure 36, for Eclipse dataset, we have the best performance using BSP_{ST+CF} compared to BSP_{ST} and BSP_{DE}. Also based on Figure 37 to Figure 41 we have the best performance using BSP_{ST+CF} compared to BSP_{ST} and BSP_{DE} for all cases but Minor severity for the Gnome dataset. Based on Figure 32 to Figure 41, we can answer RQ4 and conclude that using the categorical features including product, component and operating system in addition to stack traces improves the prediction accuracy of BSP_{ST}. The severity prediction accuracy improvement by adding categorical features ranges from 5% in Eclipse to 20% in the Gnome dataset.

Based on Figure 32 to Figure 41 we can answer RQ5 and conclude that BSP_{ST+CF} performs better than a random classifier, except for the Critical severity level of the Gnome dataset for the same reasons we explained in RQ2.

The results shown in Figure 32 to Figure 41 confirm that stack traces could be used to predict the severity of bugs with higher accuracy than the description. Furthermore, we can conclude that adding product,

component, and operating system categorical features to stack traces increases the severity prediction accuracy.

6.4.7.4. Statistical Significance of the Results

We used two-tailed Mann-Whitney test to further assess the statistical significance of the difference between the results of the approaches shown in Figure 32 to Figure 41. More precisely, we compare the F-measure of BSP_{ST+CF} to the F-measure of BSP_{ST} and also the F-measure of BSP_{ST} to the F-measure of BSP_{DE}. We consider the difference between two sets of F-measures to be statistically significant if the significance level (p-value) is less than 0.05.

Furthermore, we used Cliff's non-parametric effect size measure, which shows the magnitude of the effect size of the difference between the two sets of F-measures. The effect size estimates the probability that a value chosen from one group is statistically higher than a value chosen from another group [GK05, MRL11]. We want to calculate Cliff's effect size to estimate the probability that one F-measure obtained from one of our approaches is statistically higher than an F-measure of another approach. Cliff's effect size (*d*) is calculated as follows [GK05, MRL11]:

$$\text{Cliff's effect size} = \frac{\#(x_1 > x_2) - \#(x_1 < x_2)}{n_1 * n_2} \quad (21)$$

In (21), x_1 and x_2 are F-measure values within each group (each approach), # indicates the number of times values of one group is higher or lower than values of other group and n_1 and n_2 are the size of each approach result. More precisely, let's define d_{ij} as follows [GK05, MRL11]:

$$d_{ij} = \begin{cases} +1 \; if \; F - measure_i \; from \; first \; approach \; > \; F - measure_j \; from \; second \; approach \\ -1 \; if \; F - measure_i \; from \; first \; approach < \; F - measure_j \; from \; second \; approach \\ 0 \; if \; F - measure_i \; from \; first \; approach = \; F - measure_j \; from \; second \; approach \end{cases} \quad (22)$$

Based on Equation (22), we can define Cliff's effect size as follows [GK05, MRL11]:

$$\text{Cliff's effect size} = \frac{\sum_i \sum_j d_{ij}}{n_1 * n_2} \quad (23)$$

Cliff's effect size ranges from [-1, +1]. Cliff's effect size of +1 indicates that all the values of the first group are larger than the second group and -1 shows that all values of the first group are smaller than the

second group. Considering Cliff's effect size as *d*, effect size is small when $0.147 \le d < 0.33$, medium for $0.33 \le d < 0.474$ and large for $d \ge 0.474$ [GK05, MRL11]:

Table 20 Mann-Whitney test significance level and Cliff's effect size of the approach with stack traces and categorical features and an approach which uses stack traces alone for the Eclipse dataset

	Critical	Blocker	Major	Minor	Trivial
Two tailed Mann-Whitney test significance level	0.045	0.007	0.02	0.0001	0.0007
Cliff's effect size	0.59	0.72	0.62	0.98	0.90

Table 21 Mann-Whitney test significance level and Cliff's effect size of the approach with stack traces and categorical features and an approach which uses stack traces alone for the Gnome dataset

	Critical	Blocker	Major	Minor	Trivial
Two tailed Mann-Whitney test significance level	0.0001	0.00018	0.00028	0.00018	0.00018
Cliff's effect size	1	1	0.97	1	1

Table 20 and Table 21 show the result of Mann-Whitney test significance level and Cliff's effect size test between BSP_{ST+CF} and BSP_{ST} for Eclipse and Gnome, respectively. For both datasets and all severity levels, the difference of F-measure values of the two approaches is statistically significant with a p-value < 0.05. Hence, we can conclude that the difference between the F-measures of BSP_{ST+CF} and BSP_{ST} is statistically significant for both datasets.

Furthermore, since for all severity labels, Cliff's effect size of BSP_{ST+CF} compared to BSP_{ST} is more than 0.474, we can conclude that the BSP_{ST+CF} outperforms the BSP_{ST} with a large effect size for both datasets.

Table 22 Mann-Whitney test significance level and Cliff's effect size of the approach with stack traces and an approach that uses bug report descriptions for the Eclipse dataset

	Critical	Blocker	Major	Minor	Trivial
Mann-Whitney test a significance level	0.001	0.00736	0.79486	0.00168	0.72786
Cliff's effect size	0.88	0.72	-0.08	-0.84	0.1

Table 23 Mann-Whitney test significance level and Cliff's effect size of the approach that uses stack traces and an approach that uses bug report descriptions for the Gnome dataset

	Critical	Blocker	Major	Minor	Trivial
Mann-Whitney test a significance level	0.34722	0.00018	0.00018	0.00018	0.8493
Cliff's effect size	0.26	1	1	-1	0.06

Table 22 and Table 23 show the result of Mann-Whitney test significance level and Cliff's effect size between BSP_{ST} and BSP_{DE} for Eclipse and Gnome, respectively.

For the Eclipse dataset, the difference of F-measure values of the two approaches is statistically significant with a p-value < 0.05 for all severity levels, except for Major and Trivial. As shown in Table 12, since the Major severity is the majority class label, and the classifier is normally biased toward the majority class label, we have the same accuracy when using BSP_{ST} as when using BSP_{DE}. For Trivial, as shown in Table 14, since the Trivial severity level is under-sampled, we see the same effect. This is because the classifier is biased toward the majority class label.

Furthermore, for Critical and BLocker, the Cliff's effect size of BSP_{ST} compared to BSP_{DE} is more than 0.474 and BSP_{ST} outperforms BSP_{DE} with a large effect. For the Major severity level, the Mann-Whitney test and Cliff's effect size test results are consistent. Consistent with the Mann-Whitney test, Cliff's effect size for Major severity is close to zero, which shows that the difference in F-measure values of two approaches is not statistically significant. For trivial severity, consistent with the Mann-Whitney test, Cliff's effect size for Trivial severity is small. The reason is that the Trivial severity label is under-sampled. For the Minor severity level, the Mann-Whitney test shows that the difference between F-measure values of the two approaches is statistically significant and the Cliff's effect size shows that BSP_{DE} largely outperforms BSP_{ST}.

For the Gnome dataset, the difference of F-measures of the two approaches is statistically significant with a p-value < 0.05 for all severity levels, except for Critical and Trivial.

Furthermore, for all severity labels other than Critical and Trivial, Cliff's effect size is large. For Blocker and Major severity, Cliff's effects size of BSP_{ST} compared to BSP_{DE} is more than 0.474. For the Critical severity level, consistent with the Mann-Whitney test which shows that the difference of F-measure values of the two approaches is not statistically significant, Cliff's test shows the effect size is small. Also, for Trivial severity level, we have consistent results from Mann-Whitney test and Cliff's test (Mann-Whitney p-value>0.05 and Cliff's test is close to zero) which both show the difference of F-measure values of BSP_{ST} and BSP_{DE} are not statistically significant and both approaches have the same performance. For the Minor severity level, the Mann-Whitney test shows that the difference of F-measure values of the two approaches is statistically significant and Cliff's effect size shows BSP_{DE} outperforms BSP_{ST} by a large effect size.

6.4.7.5. Analysis of Lower Performance of the Proposed Approach

For Eclipse dataset in the case of Minor severity, we had lower performance when using BSP_{ST} compared to using the BSP_{DE}. We investigated the dataset to study the underlying reason.

Bug reports from Eclipse with Minor severity are mainly not from the Eclipse platform, but from the Eclipse plugins. Hence, their stack trace significantly varies from one to another. Studies [LM13] show that BSP_{ST} can detect duplicate bug reports with higher accuracy compared to the approach which uses bug report descriptions. The results presented in Figure 32 to Figure 41 confirms the hypothesis and shows that BSP_{ST} can predict bug severity with higher accuracy than BSP_{DE}. However, in the case of bugs with Minor severity, due to the variety among plugins, there are fewer bug reports with similar stack traces. Hence, we have lower severity prediction accuracy using BSP_{DE} compared to BSP_{ST}.

We also studied bugs with Minor severity and observed that many bug reports with Minor severity have categorical features available in their header or description. Since categorical information is stored in the header or description of the bug reports, when using BSP_{DE}, categorical features are being used to predict severity too. These categorical features boost the severity prediction accuracy of bug report

descriptions. For instance, bug report#296383 and bug report#313534 of Eclipse bug repository are shown in Table 24 and Table 25 respectively.

Table 24 Eclipse Bug Report #296383

Bug report field	Value
Product	EclipseLink
Component	JPA
Header	***JPA*** 2.0 server test script requires better rebuild integration with Eclipse IDE development
Description	If you are developing in eclipse while running these server tests - note that there is currently an issue with the ***JPA*** 2.0 test framework where a full build in eclipse will remove classes expected by the server test ant script. *You will see the following issue unless you do a full ant trunk build after any Eclipse IDE rebuild or clean .Do a 2nd rebuild off of trunk or ***eclipselink***.jpa.test after any IDE development.

Table 25 Eclipse Bug Report #313534

Bug report field	Value
Product	EclipseLink
Component	JPA
Header	***JPA***: entities in separate eclipse project must be explicitly listed in persistence.xml
Description	<exclude-unlisted-classes> has no effect in this configuration Configuration: (I am using an eclipse .classpath reference only) - and not generating a jar file - no MANIFEST.MF Class-Path entry - no <jarfile> element in persistence.xml - persistence.xml is in client project - no persistence.xml in entities project .classpath = <classpathentry combineaccessrules="false" kind="src" path="/org.eclipse.persistence.example.jpa.server.entities"/> <classpathentry kind="src" path="src"/> This may be expected behavior for SE PU's when we fail to use the persistence.xml element - as mormally the entities must be at the root of the classpath that contains persistence.xml <jarfile>entities.jar</jarfile>>Found: Using <exclude-unlisted-classes>false</exclude-unlisted-classes><class>org.eclipse.persistence.example.jpa.server.business.Cel l</class> [EL Config]: 2010-05-19 10:24:01.195--ServerSession(27196165)--Thread(Thread[main,5,main])--The access type for the persistent class [class org.eclipse.persistence.example.jpa.server.business.Cell] is set to [FIELD].

In both bug reports, JPA, the faulty component of the bug, appeared in the header of the bug reports. This extra information helps BSP_{DE} to outperform predictions based on BSP_{ST}. However, as shown in Table 13, if we add categorical features to stack traces, then the BSP_{ST+CF} outperforms the approach that uses only BSP_{DE} .

For the Gnome dataset, the only case that the BSP_{ST+CF} is outperformed by the BSP_{DE} is when the approach is predicting Minor severity. We investigated the reason and observed that the descriptions of bug reports with Minor severity contain more structured information than categorical features of the bug reports in Gnome Bugzilla. For example, as shown in Table 26, Gnome bug report#273727, which is a bug with Minor severity, not only contains all important categorical features in its description but also contains information regarding the bug's package and the step to reproduce the bug. Moreover, in addition to categorical features, sometimes the source code is copied in the description of the bug report with Minor severity. For example, Gnome bug report# 532680, shown in Table 27, contains the source code of the bug. This information causes the approach which uses descriptions to outperform the approach which uses stack traces when predicting the Minor severity.

6.5. Threats to Validity

In this Section, we explain the threat to the external validity, internal validity and construct validity of our approach.

6.5.1. Threats to External Validity

We evaluated both proposed approaches using two well-known open-source datasets. We also used the history of bug reports in those datasets from the time of the creation of bug reports in bug repositories until 2015. While the results of our experiments show that leveraging categorical features improves the severity prediction accuracy, in order to generalize these results, our approach needs to be tested on a bigger pool of datasets.

Table 26 Gnome Bug Report#273727

<table>
<tr><td>Bug report field</td><td>Value</td></tr>
<tr><td>Product</td><td>evolution</td></tr>
<tr><td>Component</td><td>Shell</td></tr>
<tr><td>Header</td><td>sanity check for e-d-s on start</td></tr>
<tr><td>Description</td><td>Descriptionorogor 2005-03-15 19:41:19 UTC
Distribution: Gentoo Base System version 1.4.16
Package: Evolution
Priority: Normal
Version: GNOME2.8.1 2.0.3
Gnome-Distributor: Gentoo Linux
Synopsis: Random UI crash
Bugzilla-Product: Evolution
Bugzilla-Component: Calendar
Bugzilla-Version: 2.0.3
BugBuddy-GnomeVersion: 2.0 (2.8.1)
Description:
Description of the crash:
It just exit
Steps to reproduce the crash:
1. Do random stuff , preferably around the contact list
2. wait
3. play with it again
4 Expected Results: than it doesn t exit. How often does this happen?
maybe average session is 15 minutes (depends how mutch you use the ui
)</td></tr>
</table>

Table 27 Gnome Bug Report #532680

<table>
<tr><th>Bug report field</th><th>Value</th></tr>
<tr><td>Product</td><td>GIMP</td></tr>
<tr><td>Component</td><td>Plugins</td></tr>
<tr><td>Header</td><td>help-browser segfaults on 64bit systems</td></tr>
<tr><td>Description</td><td>It was the latest Ubuntu HH version of the libgtk2.0-0 package.
% dpkg -p libgtk2.0-0
Package: libgtk2.0-0
Architecture: amd64
Source: gtk+2.0
Version: 2.12.9-3ubuntu3
Now it is immediately clear what went wrong:
_gdk_x11_convert_to_format has parameter src_buf==NULL
This shows that we have a pixmap that is non-NULL, with pixmap->pixels that is NULL.
This pixmap is found in the cache. If I insert the condition
&& gdk_pixbuf_get_pixels(icon->pixbuf) != NULL

in gtk+-2.12.9/gtk/gtkiconfactory.c around line 2445:

@@ -2441,7 +2441,8 @@
if (icon->style == style &&
icon->direction == direction &&
icon->state == state &&
- (size == (GtkIconSize)-1 || icon->size == size))
+ (size == (GtkIconSize)-1 || icon->size == size) &&
+ gdk_pixbuf_get_pixels(icon->pixbuf) != NULL)
{
if (prev)
{
then the segfault goes away. I have not investigated further what the real cause of the problems is.</td></tr>
</table>

Moreover, in Eclipse, stack traces are stored in the description of the bug reports and are optional. Only 10% of Eclipse bug reports contain stack traces. Even though Eclipse established an automatic stack trace collection system since 2015, the stack traces are not publicly available yet. On the other hand, in the Gnome dataset used in this chapter, in which same as the Eclipse bug repository stack traces are stored in the description, 45% of bug reports contain stack traces.

We evaluated our approach using the severity labels that are reported from users and further revised and adjusted (if need be) by the triagers. Errors may occur when reporting or adjusting severity levels, which can impact our analysis.

6.5.2. Threats to Internal Validity

One of the sources of internal threats is the misclassification function used in our approach. In our study, we used Equation (15) to calculate the misclassification cost. This misclassification cost equation was derived based on heuristics. While the results obtained using Equation (15) are convincing, using a more optimized approach for deriving misclassification cost of each severity label could further improve the severity prediction accuracy.

We set a threshold of ten to be our upper bound for the misclassification cost based on the criteria we explained in Section 6.4.2. A different threshold may have yield different results.

The regular expression used for extracting stack traces from bug report descriptions may have missed some stack traces or some functions in the stack traces. This could reduce the accuracy of the severity prediction approach. Using a different regular expression may improve the results.

6.6. Conclusion

In this chapter, we proposed two new severity prediction approaches. A new approach which uses stack trace to predict severity of bugs and also an approach which combines the use of stack traces and categorical features. The first approach uses stack traces only and the second approach uses a linear

combination of stack trace similarity and categorical features similarity to predict the severity of bugs. Since the dataset (i.e., the labels in bug report repositories) is normally unbalanced, we adopted a cost-sensitive K-nearest neighbour approach to predict the severity.

We used two open-source and popular datasets (Eclipse and Gnome bug repositories) to evaluate our approach. The result showed that in both cases the accuracy of predicting the severity of bugs is higher using BSP_{ST} compared to BSP_{DE}. Moreover, adding categorical features and using the linear combination of stack traces and categorical features similarity can further improve the severity prediction accuracy for both datasets.

Chapter 7

Automatic Prediction of Bug Report Faulty product and component fields

When a bug report is submitted, it is examined by a triaging team with the objective of redirecting it to the development team, which is expert in the affected software component. To achieve this task, triagers rely heavily on the information provided in the bug reports. It has been shown that it is common for users to enter incorrect bug report fields [BJSWPZ08, LD13]. Xia et al. [XLSW16] showed that 80% of bug reports have their fields reassigned several times after they have already been submitted to developers to be fixed. Battenberg et al. [BJSWPZ08] showed that there is an important gap between what users provide as input and what developers need to fix the bug. They explained that since end-users do not usually have technical knowledge about the system, it is very difficult for them to set bug report fields properly. As a result, many bug reports take a considerable amount of time to be fixed.

Among the reassigned bug report fields, the component and product fields are the ones that are reassigned the most, as shown by Lamkanfi et al. [LD13], Xie et al. [XZM13], Guo et al. [GZNM11], and Sureka et al. [S12], which explains the growing number of studies that aim at predicting these fields. Sureka et al. [S12] have proposed an approach based on bug report descriptions to predict faulty components. Wang et al. [WZLLW12] have compared the effectiveness of different machine learning techniques to predict faulty components. Existing studies, however, rely on bug report descriptions as the main features for classification.

As a motivating example, consider the history of Eclipse bug report #215679, shown in Figure 42. The report was first assigned to a developer to be fixed on January 17th, 2008. The component and product fields were first changed from UI and Platform to PHP Explorer View and PDT, respectively. After 16 months, these fields were changed again to Common and DLTK, which were the correct fields. Clearly there is a need to develop techniques and tools that can predict the correct product and component

fields at the time of submission of a bug report. Such techniques can provide tremendous help to triagers in processing the bug reports.

Who	When	What	Removed	Added
remy.suen	2008-01-17 14:05:31 EST	CC		remy.suen
pwebster	2008-01-17 15:04:14 EST	Assignee	Platform-UI-Inbox	php.ui-inbox
		Component	UI	PHP Explorer View
		Product	Platform	PDT
		Version	3.4	unspecified
spektom	2009-05-08 15:30:51 EDT	CC		spektom
		Component	PHP Explorer & Projects management	Common
		Product	PDT	DLTK
		Version	unspecified	1.0
spektom	2009-05-08 15:35:34 EDT	Status	NEW	RESOLVED
		Resolution	---	FIXED

Figure 42 Eclipse bug report #215679 history information

In this chapter, we propose an approach to automatically predict the component and product fields of bug reports based on a combination of stack traces and categorical information (system version, severity, and platform). Our approach relies on mining historical bug reports in order to predict faulty component and product fields of new incoming bugs. More precisely, we map stack traces of historical bug reports to term vectors, weighed using TF-IDF (term frequency-inverse document frequency). The term vectors, together with categorical information, are then used to predict the faulty component and product fields associated with an incoming bug report.

We show the effectiveness of our approach by applying it to Eclipse bug reports submitted between October 2001 to February 2015 and to Gnome bug reports submitted between February 1997 and August 2015. Moreover, we show that our approach outperforms existing studies that rely mainly on bug report descriptions [S12]. To the best of our knowledge, this is the first time that stack traces and categorical features are used to predict faulty product and component fields of bug reports.

This chapter is organized as follows: We discuss the features used for product and component prediction in Section 7.1. The proposed approach is explained in Section 7.2. We discuss the evaluation and results in Section 7.3. Threats to validity are presented in Section 7.4. The conclusion is presented in Section 7.5.

7.1. Bug Report Features

For this study, we used two types of features to train our classification algorithm: stack traces and bug report categorical information. A stack trace ideally represents a log of the information found in the stack memory at the time of the crash, however, in some cases, this information is trimmed.

In addition to stack traces, we used the following categorical information: version, operating system, and severity. The motivation behind using these categorical attributes comes from the work of Xia et al. [XLSW16], who showed that using categorical features (called meta-features in their study) improves the prediction accuracy of bug report fields that will most likely be reassigned. Categorical features are also used in Chapter 5 [SLKJ11] and Chapter 6 [SHH19] as additional features to detect duplicate bug reports and predict the severity of bugs respectively.

Furthermore, in the previous chapter, we showed that stack traces and categorical information are better features than bug report descriptions for predicting the severity of bugs, another important bug report field. This motivated us to apply a similar approach for predicting bug report product and component fields. We summarize the steps of the training and testing phases of our approach in Section 7.2.

7.2. The Proposed Approach

In this Section, we propose a faulty product and component prediction approach that uses stack traces and categorical information. We predict faulty product and component fields of a bug report using its K-nearest neighbours bug reports. We use linear combination of similarity of stack traces and categorical features of bug reports to measure the similarity of bug reports.

7.2.1. Predicting Bug Report Faulty Product and Component Fields

In this chapter, we follow an approach based on the linear combinations of stack traces and categorical features similarity to calculate the similarity of bug reports and predict faulty product and component fields. Given two bug reports (B_1, B_2) the combined similarity is calculated as follows:

$$\text{SIM}(B_1, B_2) = \sum_{i=1}^{4} w_i * feature_i \quad (24)$$

Where $feature_1, feature_2$, $feature_3$ and $feature_4$ are defined as follows:

$$feature_1 = Similarity\ of\ stack\ traces$$

$$feature_2 = \begin{cases} 1 & if\ B1.version = B2.version \\ 0 & otherwise \end{cases}$$

$$feature_3 = \begin{cases} 1 & if\ B1.severity = B2.severity \\ 0 & otherwise \end{cases}$$

$$feature_4 = \begin{cases} 1 & if\ B1.platform = B2.platform \\ 0 & otherwise \end{cases}$$

In Equation (24) similarity of categorical features is one if they are the same and zero if they are not. The SIM function in Equation (24) contains four free parameters (w_1, w_2, w_3, w_4) that represent the weights we assign to each feature. These weights are adjusted in a separate training phase. Our training dataset format is similar to the one presented by Sun et al. [SLKJ11]. It contains triples in the form of (*q, rel, irr*), where *q* is the incoming bug report, *rel* is a bug report with the same product or component and *irr* is a bug report with a different product or component. The method used to create the training set is shown in Figure 43.

We need to define a cost function to optimize the free parameters (w_1, w_2, w_3, w_4) based on our training set. We used the RankNet Cost function (RNC) of Equation (7) as our cost function.

We need to find the values of each of the four free parameters (w_1, w_2, w_3, w_4) that minimizes the cost function of Equation (7). The cost function is minimized when the similarity of bug reports with the same product or component is maximized, and the similarity of bug reports with different components or products is minimized. We use gradient descent as shown in Figure 16, provided by Sun et al. [SLKJ11], to minimize the above cost function.

Method inputs:
N>0 size of TS
$G=\{G_1, G_2, G_3, G_4, \dots, G_n\}$
$G_i = bug\ reports\ of\ product_i\ \ or\ component_i$
Method output:
TS (Training set).

Initialize *training set to be empty (TS =* $\emptyset$*)*
For each Group G_i in the repository **do**
  R= all bug reports in Group G_i
  For each report *q* in *R* **do**
    For each report *rel* in *R*-{*q*} **do**
      For *i=1* to *N* **do**
        Randomly choose a report *irr* out of *R*
        TS=TS U {(q,rel,irr)}
      End For
    End For each
  End For each
End For each
Return *TS*

Figure 43 Training dataset

The optimization algorithm adjusts each free parameter x in each iteration according to coefficient Ƞ and partial derivative of RNC with respect to each free parameter x. Then the four free parameters (w_1, w_2, w_3, w_4) are used to calculate the similarity of each incoming bug report to all the previous bug reports in the dataset to predict the faulty product and component fields using a cost-sensitive K-nearest neighbour. We explain the method used to extract stack traces and categorical features in Section 7.2.2. Next, we explain the method used for calculating the similarity of stack traces and the K nearest neighbour method for predicting faulty product and component fields in Section 7.2.3.

7.2.2. Bug Features Extraction (Stack Trace Extraction and Categorical Features Extractions)

We use the regular expression introduced in Section 3.2 to extract stack traces from Gnome bug report descriptions. We also used the regular expression by Lerch et al. [LM13] introduced in Section 3.1 to extract stack traces from Eclipse bug reports descriptions.

We implemented a custom parser to extract bug report categorical features for Eclipse and Gnome. Eclipse and Gnome both use Bugzilla, each bug report is exported as XML and the parser is used to parse the XML and extract categorical features.

7.2.3. KNN Classifier for Faulty Component and Product Classification

For each incoming bug report, we extract the corresponding stack traces, build the weighed feature vector and compare it to feature vectors of all previous bug reports in the dataset. The calculated similarities are used to build a list that shows how similar the current bug report stack traces is to all the previous stack traces of all bug reports in the dataset. These similarities are combined with the similarity of categorical features. Then we use the K-Nearest Neighbour (KNN) algorithm to retrieve the most similar bug reports to the incoming bug report. We use Equation (24) to calculate and return similar bug reports in the KNN method.

Next, we chose K nearest bug reports from the dataset and chose the label of the bug report B_i based on majority voting based on Equation (10). The score function of Equation (10) is defined as Equation (11). Based on Equation (10) and (11) the label with the highest frequency of occurrence among the K returned labels is considered as the output label.

Since the bug reports that are closer to the incoming bug report $\mathrm{B_i}$ must have more impact on choosing the incoming bug report $\mathrm{B_i}$ product or component label, we need to give more weight to the bug reports that are closer to the incoming bug report.

If we assume the distance of the closest bug report in the sorted list of K nearest instances as $dist_1$ and the distance of the farthest bug report as $dist_k$, we weigh each of the bug reports in the list of nearest neighbours returned using the Equation (12). Next, we incorporate the calculate weights from Equation (12) in Equation (10) and defined the Equation (13) which assigns scores to each faulty product and component class considering the distance of k nearest neighbour returned bug reports.

Ideally, there are equal samples for each label in the training set, however, in large software systems such as Eclipse and Gnome, some products or components have fewer bug reports in the bug tracking system, which results in an unbalanced distribution of labels.

Classifiers tend to increase overall accuracy if trained on an unbalanced dataset, which will result in a bias towards the majority class labels. We use cost-sensitive learning [ZM03] to overcome the unbalance dataset problem.

At the first step, we need to convert the output of the K-nearest neighbour classifier of Equation (13) to probabilities. We defined probability as the value of class label weight divided by the sum of all class label weights based on Equation (14)

We set the misclassification cost in a cost matrix that corresponds to the confusion matrix [LD13] of Figure 29. We need to assign high misclassification costs to the false positive and false negative in the confusion matrix since they are considered as wrong classification results. Furthermore, true positive and true negative show correct classification results and are favourable conditions, so we set the misclassification cost of them to be zero. Considering that the faulty product and component prediction problem is a multiclass classification scenario, we extend the same rationale to the multiclass classification case. In the multiclass classification scenario, the diagonal of the confusion matrix is representative of true positives and true negatives, so we set the misclassification cost on the diagonal of the confusion matrix to zero. In addition, the cost of false positives and false negatives are chosen based on the classification scenario. In this chapter, we use Equation (15) which defines misclassification cost to be reciprocal to the number of instances of each class label divided by the number of instances of majority class label to build the cost matrix.

After building the cost matrix according to Equation (15), we can calculate the classification cost of each class label for each instance using the cost matrix and the probability of the instance in the test set

belonging to each class label based on Equation (14) using new cost-sensitive classifier [ZM03] of Equation (16). Next, the class label with the lowest classification cost is chosen as the label for instance.

In this chapter, we use Equation (15) to calculate the misclassification cost of each label. We choose a threshold of ten as the maximum value for misclassification cost to avoid very high misclassification cost, which may happen due to the very limited number of instances of some minority class labels. Determining the optimal value of the threshold is beyond the scope of our study.

7.2.4. Overall Approach

Since we need to understand the importance of similarity of stack traces compared to categorical features, we chose to use a linear model with a gradient descent optimizer. In our approach with incoming of each bug report, its label is predicted considering all the previous bug reports in the dataset. To make our approach easily deployable in practice, we chose to use K-nearest neighbour for classification in the mentioned setting.

We explain the Training and Testing phase of our online severity prediction method in this Section. Figure 44 shows the overall approach.

7.2.4.1. Training

Our training phase is shown in Figure 44. In our training phase, we extract stack traces from bug report descriptions using the regular expression introduced in Section 7.2.2. We also extract categorical features using the same parser introduced in Section 7.2.2. Stack traces are converted to feature vectors and weighed using TF-IDF, explained in Section 5.1. Next, stack traces are compared using cosine similarity of Equation (6). We use linear combination (Equation (24)) of the corresponding stack traces and categorical features similarity to take categorical features into consideration in the final similarity results. The linear model is trained using the training dataset build according to Figure 43 and optimized using the gradient descent. The tuned coefficients (w_1, w_2, w_3, w_4) are then used the testing phase.

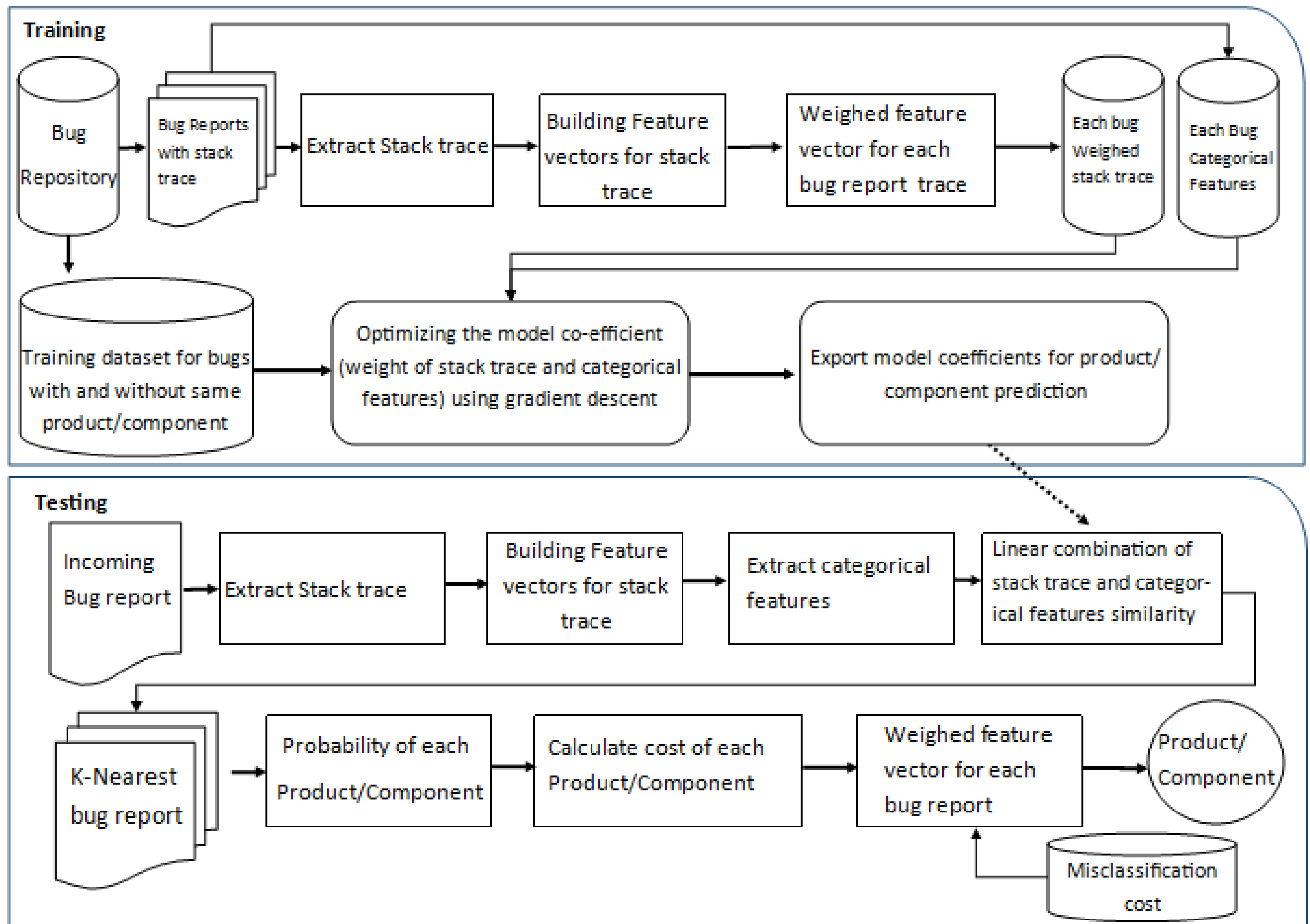

Figure 44 Overall approach

7.2.4.2. Testing

Our online faulty product and component prediction approach is depicted in Figure 44. With the incoming of each bug report to the system, its stack trace is extracted using regular expression introduced in Section 7.2.2. The similarity of the bug report to all the previous bug reports is then calculated using the linear combination of stack traces and categorical features similarity of Equation (24) based on the coefficient output of the training phase. After calculating top K nearest bug reports to the incoming bug report, the faulty product and component fields of the incoming bug report is predicted.

7.3. Evaluation

In this Section, we evaluate the accuracy of predicting faulty product and component fields using the proposed approach and compare it to the Sureka et al. [S12] approach. We conduct experiments to answer the following research questions:

> **RQ1**. Can stack traces and categorical features be used to predict the product field of a bug report, and if so, what would be the accuracy and how does it compare to the use of bug report descriptions?
>
> **RQ2**. Can stack traces and categorical features be used to predict the component field of a bug report, and if so, what would be the accuracy and how does it compare to the use of bug report descriptions?
>
> **RQ3**. How does our approach compare to a random classifier?

7.3.1. Experimental Setup

In our experiments, we compared our approach (Figure 44) to the approach that uses the description of bug reports and random approach. It is important to mention that in Eclipse and Gnome, stack traces are embedded in the description. In this Section, we describe the dataset used in the experiments and provide statistical analysis regarding the dataset.

7.3.1.1.1. The dataset

In this chapter, we used bug reports extracted from the two large open-source software projects: Eclipse and Gnome. These systems have their bug reports open and accessible to researchers and have been widely used in the literature [AADPKG08, LDGG10, LDSV11, TLS12, YZL14, ZYLC15].

Eclipse contains a comprehensive set of bug reports with their faulty product and component fields. As a universal tool platform, the Eclipse project has been very active with new products and components

added on a daily, weekly and monthly basis. Gnome is a collection of Unix-based projects. Since it is an open-source project, developers contribute to it daily and it consists of many products and components.

7.3.1.1.2. Dataset Setup

In this chapter, we use 10% of our dataset is used for training the linear model of Equation (24). The rest of the 90% is used as the testing set. In our testing set, we use our online classification model to predict faulty product and component fields of the bug reports. With the incoming of each bug report its similarity to all the previous bug reports in the dataset is calculated. We use cost-sensitive K-nearest neighbour to predict the faulty product and component fields of the incoming bug report by comparing it to all the previous bug reports.

Since bug reports follow a temporal order, we compare each bug report to only previous bug reports in the dataset. We cannot use K-fold validation since we cannot train the model on the future data and use it to predict the faulty product and component fields of the current data.

7.3.1.1.3. Dataset Analysis

Eclipse platform can be extended by means of plugins. Each plugin that is separated from Eclipse has its own sets of products and components. In this work, we have focused only on Eclipse products and components and ignored the plugins. Eclipse has five products: Platform, JDT, PDE, Equinox and E4, each of which contains a set of components. In Eclipse, stack traces are not provided separately; they are indeed pieces of the description of bug reports. The list of Eclipse products and the number of components of each product are shown in Table 28.

Table 28 Products and components in Eclipse dataset

Product name	**Number of components**
Platform	17
JDT	5
PDE	4
Equinox	9
E4	3
Total	38

Although Gnome uses the same bug tracking system as Eclipse, it is structured slightly differently. Most of Gnome products are not broken down into components. Instead, one upper layer of abstraction is defined (Classification), in which each bug has a class, and each class has its own set of products. In this work, we considered bug classes in Gnome to be the same as bug products in Eclipse and bug products in Gnome to be the same as bug components in Eclipse.

Table 29 Products and components in Gnome dataset

Product name	Number of components
Platform	12
Core	9
Applications	12
Infrastructure	2
Bindings	7
Other	18
Deprecated	20
Total	80

The total number of Eclipse bug reports is 193,177, but only 19,458 (10%) have stack traces. This percentage results from the fact that up to 2015, stack traces had to be appended manually by users. The automatic submission of stack traces to the Eclipse bug report repository has been made possible by Eclipse at the end of 2015. So far, these stack traces have not been made publicly available. In the case of Gnome, the total number of bug reports is 629,549, among which 201,580 (32%) have stack traces. The list of Gnome products and the number of components of each product are shown in Table 29.

The sparsity of stack traces is a limitation of our approach. Nevertheless, we believe that it is still important to investigate the use of stack traces, especially that there is a recognized need to have stack traces for debugging, bug reproduction, and other software maintenance tasks. We should expect to see more bug reporting systems collect stack traces automatically whenever a bug report is submitted. Detailed characteristics of our datasets are presented in Table 30.

Table 30 Characteristics of the datasets

Data	Eclipse	Gnome
Total number of bug reports	192,998	697,800
Total number of bug reports with stack traces	19,458 (10%)	201,580 (32%)

7.3.2. Predicting Faulty Product and Component Fields of Bug Reports Using Description

We tokenize words from Eclipse and Gnome description using space and new line character. We use the raw distinct extracted words to create the feature vector. Feature vector corresponding to each bug report is weighed using TF-IDF, explained in Section 5.1. After ordering bug reports based on their creation dates, we compare incoming bug reports to all the previous bug reports in the dataset. We use the cost-sensitive classifier introduced in Section 6.4.2 to predict the faulty product and component fields of the bug reports.

7.3.3. Predicting Faulty Product and Component Fields of Bug Reports Using Random Approach

We compare our proposed approach with the approach which predicts faulty product and component fields randomly in proportion to each class label. If we assume that there are N products and the number of bug reports belonging to each product is $B_{s_1} \dots \dots\ B_{s_N}$ then the accuracy of the approach which predicts product randomly for the label $product_i(S_i)$ is calculated using (17). For each product, we have predicted its components using the same proposed random approach.

7.3.4. Faulty Product and Component Prediction Approaches Setup

We used the cost-sensitive classification approach introduced in Section 6.4.2 to overcome the unbalanced dataset distribution problem. We assessed the performance of all models by varying the returned list size 1 to 10. For training the linear model of Equation (24) we initialized from a normal

distribution with zero mean and a standard deviation of 0.1. We also used 0.001 as the learning rate and trained the model on the first 10% of the dataset.

7.3.5. Evaluation Metrics

We used precision, recall, and F-measure to assess the effectiveness of our approach. These metrics are widely used in the literature [LD13, XLSW16] to evaluate the accuracy of a classifier. We defined product (component) prediction precision as the ratio of bug reports for which we have correctly predicted the product field p_L (component field C_L) (true positives) to the total number of bug reports for which we predicted product field p_L (component field C_L) (true positives + false positives). We calculated the precision for each product and component label separately.

$$Precision\ (p_L) = \frac{\#\ of\ bugs\ correctly\ predicted\ with\ label(p_{L/CL})}{\#\ of bugs\ predicted\ to\ have\ label(p_{L/CL})} \quad (25)$$

The recall is defined as the ratio of bug reports for which we have correctly predicted product field p_L (component field C_L) (true positives) to the total number of bug reports which actually have the product label p_L (component field C_L) (true positives + false negative) .

$$Recall\ (P_L) \ = \frac{\#\ of\ bugs\ correctly\ predicted\ with\ label\ (P_{L/CL})}{\#\ of\ bugs\ actually\ having\ label\ (P_L/CL)} \ (26)$$

We built the confusion matrix separately for each product or component field. We combined precision and recall values and presented them as one value, F-measure of Equation (20).

To compare the results of our approach to Sureka's [S12] approach, which uses bug report descriptions, and also the random approach, we measured the improvement achieved by one method over the other. More precisely, if we denote the F-measure of Sureka's [S12] approach as $F_measure_{Text}$ and the F-measure of our approach by $F_measure_{Stacktrace}$, we calculate the improvement as follows:

$$\text{Improvement} = \frac{F{-}measure_{Stacktrace} - F{-}measure_{Text}}{F{-}measure_{Text}} \quad (27)$$

Because we have a large number of components for each product and that the precision and recall must be calculated separately for each component, we used the macro-average precision to show the average precision of the components of each product. If we denote precision of the first component as P_{C1} and the nth component precision as P_{Cn}, we can use the Equation provided by Manning et al. [MRS08] to calculate the macro-average precision:

$$Macro_{average}precision = \frac{P_{C1} + P_{C2} + \cdots . + P_{Cn}}{n} \quad (28)$$

Similarly, if we denote recall of the first component as R_{C1} and the nth component recall as R_{Cn}, the macro-average recall is calculated using the following Equation [MRS08].

$$Macro_{average}recall = \frac{R_{C1} + R_{C2} + \cdots . + R_{Cn}}{n} \quad (29)$$

In our experiments, we varied the value of the K (size of the similar bug reports returned) from 1 to 10, and we captured the best accuracy of the approaches. Next, we use the captured accuracies to compare approaches.

7.3.6. Evaluation Results

We have presented the results of our experiments by focusing on research questions RQ1 to RQ3, followed by a discussion Section. For simplicity reason, we use the notation BRFPst+cat to refer to our approach for predicting bug report fields using stack traces and categorical features. We also use BRFPdesc to refer to the approach that uses bug report descriptions. The approach that uses bug report descriptions predicts the faulty product and component of bug reports using the description of bug reports, including stack traces.

RQ1. Can stack traces and categorical features be used to predict the product field of a bug report, and if so, what would be the accuracy and how does it compare to the use of bug report descriptions?

Table 31 and Table 32 show the precision, recall and F-measure of BRFPst+cat to predict the product field of bug reports in Eclipse and Gnome datasets. In our experiments, we have varied K from 1 to 10 and recorded K, which provides the best accuracy.

When applied to Eclipse products, the results show that our approach, BRFPst+cat, predicts faulty products with an average F-measure of 60%. The average precision and recall are 58% and 62%, respectively. For Gnome, we applied our approach to seven products and predicted faulty products with an average precision of 74% and an average recall of 67%. The average F-measure is 70%.

Table 31 shows the average F-measure improvement of predicting faulty products using BRFPst+cat for Eclipse compared to the use of BRFPst, Sureka's [S12] approach. Based on the results of Table 31, we have improved the average F-measure from almost 5% to 143.1% for different products. The average F-measure improvement rate over all Eclipse products is 46%.

Table 32 shows that the average F-measure improvement of predicting faulty products using our approach compared to BRFPdesc for Gnome. The results show an improvement ranging from almost 4% to 174.44% for different products. On average, using stack traces and categorical features improves accuracy by 41% across all products in Gnome.

By answering RQ1, we can conclude that the use of stack traces and categorical features (system version, severity, and platform) provides better accuracy in predicting bug report product fields compared to the use of bug report descriptions.

RQ2. Can stack traces and categorical features (system version, severity, and platform) be used to predict the component field of a bug report? If so, what would be the prediction accuracy and how does it compare to the use of bug report descriptions?

Table 33 shows the macro average F-measure improvement to predict bug reports component field using BRFPst+cat compared to BRFPdesc for Eclipse. Based on the results, the improvement ranges from 0% to

almost 42%. Moreover, the macro average F-measure of the components of each product has improved using the proposed approach by 19% on average. Table 34 presents the results for the Gnome dataset. The improvement ranges from 14.73% to 50%. For the components of each product, the proposed approach outperforms BRFPdesc by 31% on average.

By answering RQ2, we can conclude that the use of stack traces and categorical features (system version, severity, and platform) provides better accuracy in predicting bug reports component field compared to the use of bug report descriptions.

Detailed precision, recall and F-measure of the proposed approach for predicting each component is presented in Table 35 to Table 46.

RQ3. How does our approach compare to a random classifier?

Table 31 and Table 32 show the product prediction accuracy of our approach using stack traces and categorical features compared to a random approach for Eclipse and Gnome, respectively. Our approach outperforms a random approach when predicting faulty products in Eclipse and Gnome datasets with an average improvement of 200% and 250% respectively. Similarly, as we can see in Table 33 and Table 34, our approach outperforms a random classifier for predicting faulty components by 205% and 391% for Eclipse and Gnome respectively.

7.3.6.1. Discussion

We showed that our approach outperforms Sureka's [S12] approach using bug report descriptions for 80% of the components for both datasets. We discuss in what follows, particular cases when Sureka's [S12] approach outperforms ours.

7.3.6.2. Product Prediction Accuracy

Table 31 Product prediction accuracy for Eclipse.

Product	Bug report Descriptions			Bug report Stack traces and categorical features			Random	Improvement over description	Improvement over random
	Precision	Recall	F-Measure	Precision	Recall	F-Measure	Accuracy		
Platform	46%	68%	54.80%	55%	68%	60%	50.2%	9.50%	19.5%
JDT	47%	68%	55.50%	69%	68%	68.40%	30.1%	23.20%	127.2%
PDE	27%	23%	24.80%	50%	76%	60.30%	9%	143.10%	570%
Equinox	34%	37%	35.40%	60%	48%	53.30%	7.9%	50.50%	574.6%
E4	55%	47%	50.60%	56%	50%	52.80%	2.7%	4.34%	1855.5%
AVERAGE	42%	49%	44%	58%	62%	60%	20%	46%	200%

Table 32 Product prediction accuracy for Gnome

Product	Bug report Description			Bug report Stack traces and categorical features			Random	Improvement over description model	Improvement over random
	Precision	Recall	F-Measure	Precision	Recall	F-Measure	Accuracy		
Deprecated	71%	75%	72.94%	81%	77%	78.94%	15.8%	8.23%	399.62%
Core	65%	73%	68.76%	70%	73%	71.46%	35.7%	3.93%	100.17%
Other	60%	65%	62.4%	68%	68%	68%	16.2%	8.97%	319.75%
Platform	69%	38%	49%	92%	72%	80.78%	2.3%	64.83%	3412.17%
Applications	74%	62%	67.47%	83%	72%	77.1%	29.6%	14.29%	160.47%
Infrastructure	35%	52%	41.83%	46%	47%	46.49%	0.1%	11.13%	46390.00%
Bindings	36%	18%	24%	78%	57%	65.86%	0.07%	174.44%	93985.71%
AVERAGE	59%	55%	55%	74%	67%	70%	20%	41%	250.00%

7.3.6.3. Macro Component Prediction Accuracy

Table 33 Component prediction accuracy for Eclipse

Product	Bug report Descriptions			Bug report Stack Traces with categorical features			Random	Improvement over description model	Improvement over random
	Precision	Recall	F-Measure	Precision	Recall	F-Measure	Accuracy		
Platform	35%	29%	31.71%	50%	41%	45.05%	5.8%	42.04%	676%
JDT	52%	49%	50.45%	67%	64%	65.46%	20%	29.74%	227%
PDE	61%	56%	58.39%	68%	60%	63.75%	25%	9%	155%
Equinox	55%	37%	44.23%	63%	43%	51.11%	11%	15.53%	364%
E4	73%	58%	64.64%	78%	56%	65.19%	33%	0%	97%
AVERAGE	55%	46%	50%	65%	53%	58%	18.96%	19%	205%

Table 34 Components prediction accuracy for Gnome

Product	Bug report Descriptions			Bug report Stack Traces with categorical features			Random	Improvement over description model	Improvement over random
	Precision	Recall	F-Measure	Precision	Recall	F-Measure	Accuracy		
Deprecated	59%	59%	59%	96%	71%	81.63%	5%	38.35%	1532.60%
Core	65%	41%	50.28%	73%	62%	67.05%	11%	33.35%	509.55%
Other	50%	42%	45.65%	75%	63%	68.48%	5.5%	50%	1145.09%
Platform	62%	48%	54.10%	85%	67%	74.93%	5.8%	38.51%	1191.90%
Applications	58%	47%	51.92%	68%	53%	59.57%	8.3%	14.73%	617.71%
Infrastructure	60%	53%	56.28%	72%	63%	67.20%	50%	19.40%	34.40%
Bindings	61%	55%	57.84%	78%	66%	71.50%	14.2%	23.61%	403.52%
AVERAGE	59%	49%	54%	78%	64%	70%	14.25%	31%	391.23%

7.3.6.4. Component Prediction Accuracy

Table 35 Component prediction accuracy (Eclipse Equinox Product)

Component	Bug report Descriptions			Bug report Stack Traces with categorical features			Improvement
	Precision	Recall	F-Measure	Precision	Recall	F-Measure	
Framework	52.00%	53%	52.40%	59%	49%	53.50%	2%
Incubator	58.00%	84%	68.60%	59%	88%	70.60%	2.90%
Compendium	39%	36%	37.40%	50%	29%	36.70%	-1.80%
Components	0	0	0	50%	11%	18%	18.18%
Launcher	50%	11%	18%	50%	11%	18%	0
DeviceKit	80%	23%	35.70%	43%	18%	25.30%	-29.10%
ServerSide	66%	14%	23.10%	86%	43%	57.30%	148%
P2	83%	81%	81.90%	86%	83%	84.40%	3%
Security	70%	38%	49.20%	77%	55%	64.10%	30.20%

Table 36 Component prediction accuracy (Eclipse PDE Product)

Component	Bug report Descriptions			Bug report Stack Traces with categorical features			Improvement
	Precision	Recall	F-Measure	Precision	Recall	F-Measure	
UI	87%	96%	91.20%	92%	96%	93.90%	2.96%
Build	38%	34%	35.80%	69%	47%	55.90%	56.10%
Incubator	42%	36%	38.70%	24%	28%	25.80%	-33.30%
Tools	78%	58%	66.50%	85%	68%	75.50%	13.50%

Table 37 Component prediction accuracy (Eclipse E4 Product)

Component	Bug report Descriptions			Bug report Stack Traces with categorical features			Improvement
	Precision	Recall	F-Measure	Precision	Recall	F-Measure	
Resources	42%	50%	45.60%	36%	50%	41.80%	-8.35
UI	96%	97%	96.40%	97%	96%	96.40%	0%
Tools	80%	28%	41.40%	100%	22%	36%	-13%

Table 38 Component prediction accuracy (Eclipse JDT Product)

Component	Bug report Descriptions			Bug report Stack Traces with categorical features			Improvement
	Precision	Recall	F-Measure	Precision	Recall	F-Measure	
Debug	57%	60%	58.40%	71%	79%	74.70%	27.90%
Core	66%	73%	69.30%	84%	75%	79.20%	14.20%
UI	58%	60%	58.90%	70%	76%	72.80%	23.50%
Text	42%	29%	34.30%	57%	50%	53.20%	55.10%
APT	35%	21%	26.20%	54%	39%	45.20%	72.50%

Table 39 Component prediction accuracy (Eclipse Platform Product)

Component	Bug report Descriptions			Bug report Stack Traces with categorical features			Improvement
	Precision	**Recall**	**F-Measure**	**Precision**	**Recall**	**F-Measure**	
Team	33%	37%	34.80%	38%	52%	43.90%	26.10%
Assistance	39%	25%	30.40%	67%	45%	53.80%	76.90%
UI	48%	71%	57.20%	59%	70%	64%	11.80%
Scripting	16%	20%	17.70%	8%	20%	11.40%	-35.50%
Text	31%	19%	23.50%	55%	44%	48.80%	107.65%
Resources	38%	43%	40.30%	50%	52%	50.90%	26.30%
Ant	48%	28%	35.30%	66%	50%	56.80%	60.90%
SWT	55%	39%	45.60%	63%	49%	55.10%	20.80%
Compare	24%	17%	19.90%	53%	42%	46.80%	135.10%
Debug	41%	28%	33.20%	57%	52%	54.30%	63.50%
Search	18%	13%	15%	56%	38%	45.20%	201.30%
Runtime	41%	33%	36.00%	49%	32%	38.70%	7.50%
Update	55%	45%	49.50%	69%	54%	60.50%	22.20%
Releng	56%	31%	39.90%	46%	28%	34.80%	-12.70%
WebDAV	0	0	0	34%	13%	18.80%	18.18%
CVS	39%	34%	36.30%	59%	44%	50.40%	38.80%
IDE	17%	11%	13.30%	17%	8%	10.88%	-18.19%

Table 40 Component prediction accuracy (Gnome Deprecated Product)

Product	Bug report Descriptions			Bug report Stack Traces with categorical features			Improvement
	Precision	**Recall**	**F-Measure**	**Precision**	**Recall**	**F-Measure**	
galeon	92%	95%	93.47%	88%	97%	92.28%	-1.28%
gnome-media	51%	71%	59.36%	71%	75%	72.94%	22.88%
gnome-core	60%	75%	66.66%	65%	79%	71.31%	6.98%
gtop	83%	45%	58.35%	83%	68%	74.75%	28.09%
gnome-pim	33%	30%	31.42%	59%	48%	52.93%	68.43%
gnome-games	63%	50%	55.75%	35%	26%	29.83%	-46.48%
gnome-utils	32%	38%	34.74%	53%	40%	45.59%	31.23%
scaffold	32%	30%	30.96%	58%	30%	39.54%	27.7%
GGV	50%	29%	36.70%	45%	29%	35.27%	-3.92%
GnomeICU	54%	64%	58.57%	74%	68%	70.87%	20.99%
acme	37%	40%	38.44%	42%	44%	42.97%	11.8%
gnome-themes	67%	29%	40.47%	60%	43%	50.09%	23.76%
gtkhtml2	63%	28%	38.76%	100%	33%	49.62%	28%
mergeant	50%	47%	48.45%	77%	77%	77%	58.91%
gnome-vfs	71%	19%	29.97%	71%	38%	49.50%	65.14%
bug-buddy	46%	38%	41.61%	55%	46%	50.09%	20.38%
gnome-python	75%	60%	66.66%	67%	80%	72.92%	9.39%
glade-legacy	67%	49	56.60%	100%	35%	51.85%	-8.39%
magicdev	59%	70%	64.03%	58%	59%	58.49%	-8.64%
gossip	70%	39%	50.09%	100%	50%	66.66%	33.09%

Table 41 Component prediction accuracy (Gnome Other Product)

Product	Bug report Descriptions			Bug report Stack Traces with categorical features			Improvement
	Precision	Recall	F-Measure	Precision	Recall	F-Measure	
gnome-panel	76%	94%	84.05%	85%	70%	76.77%	-8.66%
gnome-applets	60%	63%	61.46%	72%	64%	67.76%	10.25%
GIMP	92%	66%	76.86%	88%	79%	83.26%	8.32%
balsa	65%	43%	51.76%	75%	68%	71.33%	37.80%
rhythmbox	86%	71%	77.78%	78%	90%	83.57%	7.44%
Pan	56%	50%	52.83%	79%	81%	79.99%	51.41%
gthumb	74%	62%	67.47%	74%	84%	78.68%	16.61%
GnuCash	89%	65%	75.13%	93%	91%	91.99%	22.44%
metacity	65%	41%	50.28%	100%	70%	82.35%	63.78%
dia	64%	60%	61.94%	77%	65%	70.49%	13.80%
gnome-pilot	67%	66%	66.5%	89%	87%	87.99%	32.31%
gtranslator	42%	26%	32.12%	64%	37%	46.89%	45.98%
memprof	0%	0%	0%	75%	27%	39.71%	39.71%
conglomerate	50%	26%	34.21%	90%	54%	67.5%	97.31%
mlview	100%	33%	49.62%	100%	58%	73.42%	47.96%
gnome-alsamixer	50%	59%	54.13%	96%	89%	92.37%	70.64%
gnome-commander	60%	29%	39.1%	89%	38%	53.26%	36.21%
planner	25%	9%	13.24%	100%	45%	62.07%	368.8%

Table 42 Component prediction accuracy (Gnome infrastructure Product)

Product	Bug report Descriptions			Bug report Stack Traces with categorical features			Improvement
	Precision	Recall	F-Measure	Precision	Recall	F-Measure	
bugzilla.gnome.org	89%	95%	91.9%	91%	98%	94.37%	2.69%
website	29%	22%	25.02%	100%	44%	61.11%	144.24%

Table 43 Component prediction accuracy (Gnome Binding Product)

Product	Bug report Descriptions			Bug report Stack Traces with categorical features			Improvement
	Precision	Recall	F-Measure	Precision	Recall	F-Measure	
pygtk	47%	80%	59.21%	60%	80%	68.57%	15.81%
gnome-perl	50%	25%	33.33%	40%	50%	44.44%	33.33%
gtkmm	50%	13%	20.63%	29%	25%	26.85%	30.15%
pygobject	66%	62%	63.94%	97%	68%	79.95%	25.04%
java-gnome	100%	40%	57.14%	100%	60%	75%	31.26%
gjs	86%	27%	41.1%	100%	82%	90.11%	119.25%
seed	57%	40%	47.01%	88%	70%	77.97%	65.86%

Table 44 Component prediction accuracy (Gnome Platform Product)

Product	Bug report Descriptions			Bug report Stack Traces with categorical features			Improvement
	Precision	Recall	F-Measure	Precision	Recall	F-Measure	
gtk+	56%	70%	62.22%	79%	84%	81.42%	30.86%
GStreamer	71%	61%	65.62%	89%	94%	91.43%	39.33%
pango	55%	43%	48.27%	77%	73%	74.95%	55.27%
glib	21%	14%	16.8%	37%	15%	21.35%	27.08%
gdk-pixbuf	28%	15%	19.53%	75%	52%	61.42%	214.49%
libxslt	67%	32%	43.31%	72%	68%	69.94%	61.49%
evolution-data-server	76%	77%	76.5%	85%	94%	89.27%	16.69%
at-spi	48%	25%	32.88%	68%	33%	44.44%	35.16%
atk	28%	31%	29.42%	68%	55%	60.81%	106.7%
libxml2	65%	41%	50.28%	89%	51%	64.84%	28.96%
gtksourceview	9%	13%	10.64%	73%	53%	61.41%	477.16%
NetworkManager	73%	87%	79.39%	93%	84%	88.27%	11.19%

Table 45 Component prediction accuracy (Gnome Core Product)

Product	Bug report Descriptions			Bug report Stack Traces with categorical features			Improvement
	Precision	Recall	F-Measure	Precision	Recall	F-Measure	
nautilus	86%	90%	87.95%	89%	95%	91.9%	4.55%
gnome-control-center	71%	73%	71.99%	84%	79%	81.42%	12.5%
gnome-terminal	69%	84%	75.76%	86%	78%	81.8%	7.89%
system-monitor	84%	70%	76.36%	94%	86%	89.82%	18.42%
gnome-desktop	26%	21%	23.23%	4%	17%	23.86%	4.35%
vte	23%	9%	12.94%	35%	19%	24.63%	92.31%
gnome-session	31%	20%	24.31%	56%	34%	42.31%	75%
totem	56%	41%	47.34%	74%	70%	71.94%	53.19%
epiphany	94%	72%	81.54%	93%	90%	91.48%	10.98%

Table 46 Component prediction accuracy (Gnome Applications Product)

Product	Bug report Descriptions			Bug report Stack Traces with categorical features			Improvement
	Precision	Recall	F-Measure	Precision	Recall	F-Measure	
evolution	97%	100%	98.48%	97%	100%	98.48%	0%
Gnumeric	77%	60%	67.45%	82%	73%	77.24%	15%
gedit	64%	63%	63.5%	84%	68%	75.16%	19%
ekiga	89%	66%	75.79%	60%	91%	72.32%	-5%
yelp	59%	62%	60.46%	78%	74%	75.95%	27%
gnome-chess	80%	53%	63.76%	92%	73%	81.41%	27%
file-roller	52%	61%	56.14%	74%	64%	68.64%	23%
devhelp	34%	43%	37.97%	81%	57%	66.91%	76%
gconf-editor	27%	29%	27.96%	33%	19%	24.12%	-14%
seahorse	38%	59%	46.23%	76%	66%	70.65%	54%
anjuta	60%	20%	30%	100%	40%	57.14%	90%
sound-juicer	56%	48%	51.69%	78%	65%	70.91%	37%

7.3.6.4.1. Eclipse

For some cases Sureka's [S12] approach has outperformed ours (Table 35 to Table 39) in the Eclipse dataset, we found that this happens mainly when (1) the number of stack traces is small, and (2) bug report submitters provide descriptions which contain bug reproduction steps, source code information or other types of structural information.

7.3.6.4.1.1. The Number of Stack Traces

For the Eclipse product "E4" components "Resources" and "Tools", our approach achieved lower accuracy than Sureka's [S12] approach using bug report descriptions (Table 37). Our model needs a sufficient number of stack traces to generalize and make accurate predictions. When the number of stack traces is low, it is less likely to have shared methods (features) among stack traces (this is due to the uniqueness

of stack traces) compared to bug report descriptions, which contain common features (words) since they are written in natural language. Since the number of stack traces is low in these three products, the feature vector built using methods in stack traces could not properly represent bug reports in the vector space. So, the similarity of stack traces (bug reports) based on Equation (6) is zero for most of the cases, compromising the product and component fields prediction accuracy.

Table 47 Eclipse Bug Report #213234

Bug Report Field	Value
Product	Equinox
Component	Components
Header	IOexception on an invalid file path returned by FileLocator.toFileURL

Description	Hello Equinox Team
	I got an IOexception on the FileLocator.toFileURL() method when trying to resolve the installation path of a bundle.
	Here is the snippet
	public static IPath getInstallPath() {
	URL installURL = null;
	URL resolveURL = null;
	File file = null;
	if (installPath == null) {
	try {
	installURL = getInstallURL();
	if (isDebug()) {System.err.println("getInstallPath: installURL : " + installURL.toString());
	java.io.IOException: The filename, directory name, or volume label syntax is incorrect
	at java.io.WinNTFileSystem.canonicalize0(Native Method)
	at
	java.io.Win32FileSystem.canonicalize(Win32FileSystem.java:395)
	at java.io.File.getCanonicalPath(File.java:531)
	at
	com.anubex.ndt.core.Activator.getInstallPath(Activator.java:181)

For Eclipse “Equinox” in the case of the “Components” component, we have very low accuracy when using bug report description. By further investigating this component, we observed that there is a small number of bug reports for this component and these bug reports only contain stack traces and source code, and barely contain any words in the description field. An example of a bug report description for

this component is shown in Table 47. The same observation holds for the “Webdav” component of the “Platform” product.

7.3.6.4.1.2. Bug Report Submitters Provide Descriptions, Which Contain Bug Reproduction Steps or Source Code Information.

For the “Incubator” component of the Eclipse “PDE” product, Sureka’s [S12] approach using bug report descriptions yields better results than our approach. By further investigating the “Incubator” component bug reports, we observed that they contain steps to reproduce the bug embedded in the bug report description. An example of a snippet of the bug report description for this component is shown in Table 48.

The same observation holds for the “Device Kit” and “Equinox” products where most bug reports related to these products contain the bug reproduction steps in their description (see Bug Report #192746 in Table 49).

Table 48 Eclipse Bug Report#213234

Bug Report Field	**Value**
Product	PDE
Component	Incubators
Header	[api tooling] invalid thread access setting up API tooling
Description	steps: 1. open the editor for an element that will have a source tag added to it by the wizard 2. make a change to the type and do not save it 3. start the setup wizard specific example I used to reproduce: 1. get debug.ui from head 2. open FileLink and make a change, do not save 3. run the setup wizard on debug

7.3.6.4.2. Gnome

For the Gnome dataset, our approach outperforms Sureka’s [S12] approach in most cases except for predicting the components “gnome-games” and “gconf-editor”. We found out that this is due to the fact

that the bug reports of those components contain information which is mostly technical and different from the categorical information provided in the bug tracking system. For example, in Bug Report#408425 (Table 50), we have information such as the distribution of the Linux environment, the release information of the OS, memory status, etc.

The same observation holds for the “gconf-editor” component of the Gnome “Applications” product. In Bug Report#404634 (Table 51), for example, we can see technical information, which is embedded in the bug report description.

Table 49 Eclipse Bug Report#192746

Bug report field	**Value**
Product	Equinox
Component	Incubator.DeviceKi
Header	Try to create new DK project fails
Description	Steps to recreate: 1. New->Other->Device Kit->Device Kit Components->Connection 2. Connection Name = Something 3. Finish becomes enabled click to attempt to create the connection 4. Device Kit Error Found this error in the log !SESSION 2008-02-26 23:40:00.369 --------------------- eclipse.buildId=M20071023-1652 java.version=1.5.0_13 java.vendor=Apple Computer, Inc. BootLoader constants: OS=macosx, ARCH=x86, WS=carbon, NL=en_US Framework arguments: -keyring /Users/pddempse/.eclipse_keyring -showlocation Command-line arguments: -os macosx -ws carbon -arch x86 -keyring /Users/pddempse/.eclipse_keyring -consoleLog -showlocation

Table 50 bug report#408425

Bug report field	Value
Product	Deprecated
Component	gnome-games
Header	Crash while closing the window
Description	If you click the New button in the toolbar, so that the 'new game' dialog shows up, and then you close the window using the window manager close button, HEAD crashes. Distribution: Fedora Core release 6 (Zod) Gnome Release: 2.17.90 2007-02-10 (JHBuild) BugBuddy Version: 2.17.3 System: Linux 2.6.19-1.2895.fc6 #1 SMP Wed Jan 10 19:28:18 EST 2007 i686 X Vendor: The X.Org Foundation X Vendor Release: 70101000 Selinux: Enforcing Accessibility: Enabled GTK+ Theme: Clearlooks Icon Theme: gnome Memory status: size: 58736640 vsize: 58736640 resident: 24649728 share: 13119488 rss: 24649728 rss_rlim: 4294967295 CPU usage: start_time: 1171580860 rtime: 380 utime: 343 stime: 37 cutime:0 cstime: 0 timeout: 0 it_real_value: 0 frequency: 100

Table 51 Bug report#404634

Bug report field	Value
Product	Applications
Component	gconf-editor
Header	crash in Configuration Editor: Set up my hot keys
Description	Descriptionlvlo 2007-02-05 16:02:31 UTC Version: 2.16.0 What were you doing when the application crashed? Set up my hot keys Distribution: Ubuntu 6.10 (edgy) Gnome Release: 2.16.1 2006-10-02 (Ubuntu) BugBuddy Version: 2.16.0 Memory status: size: 34299904 vsize: 0 resident: 34299904 share: 0 rss: 12042240 rss_rlim: 0 CPU usage: start_time: 1170690882 rtime: 0 utime: 665 stime: 0 cutime:617 cstime: 0 timeout: 48 it_real_value: 0 frequency: 0

7.3.7. Implication and limitations

On stack traces: Our findings clearly show the importance of stack traces in predicting the product and component fields of bug reports. This confirms the need to collect stack traces whenever a bug report is submitted. Traces should not be copied and pasted in bug report descriptions, as it is the case in many bug tracking systems. Bug report tracking systems should be designed in a way that facilitates the collection and mining of stack traces. It is recognized that stack traces require storage and processing capabilities because of their size. For Mozilla products, for example, stack traces are only kept for one year because of the overhead caused by managing these traces [CSVP17]. Therefore, simply collecting traces may not be sufficient. We need to investigate better ways to structure their content by reducing noise and other elements that may not be needed to characterize the corresponding bug reports.

On bug report categorical attributes: We showed that categorical attributes, namely version, platform, and severity, enhance the prediction accuracy. The problem is that these attributes themselves may be entered incorrectly, which is a threat to validity for our approach. Our findings strengthen the need to have these attributes automatically and correctly generated. Users should never have to enter these attributes.

On the differences between Eclipse and Gnome: When predicting faulty product and component fields, we have approximately 10% more accuracy (60% for Eclipse compared to 70% in Gnome) for Gnome compared to Eclipse. Since Gnome has more stack traces than Eclipse, our approach has more data to be trained on. More training data helps the proposed approach to generalize better, which yields better accuracy. Furthermore, the proposed approach improves over the random approach for Gnome more than Eclipse (250% improvement for Gnome and 200% improvement for Eclipse). The reason is twofold. The first reason is the higher accuracy of the proposed approach for Gnome due to the presence of more stack traces in the Gnome dataset. The second reason is the number of products and components and their distribution in Gnome. Gnome has more products and components than Eclipse. The distribution of

faulty product and component labels are also more unbalanced. Based on the random approach proposed in Equation (17), existence of more class labels and higher unbalanced class labels reduce the random approach accuracy. Training a more accurate model because of the presence of more stack traces and less accuracy of the random model because of the number of class labels and their distribution results in better improvement of the proposed approach over the random model for the Gnome dataset.

7.4. Threats to Validity

Our proposed approach and the conducted experiments are subject to threats to validity that we categorize into three categories, namely, external, internal, and construct validity.

7.4.1. Threats to External Validity

Our approach is evaluated against two well-known open-source datasets. We tested our approach on all bug reports having stack traces from the beginning of the bug repository to 2015. We showed that our approach (using stack traces and categorical features) outperforms the approach that relies on bug report description to predict the product and component fields of bug reports. We need to apply our model to more datasets to see if it outperforms bug report descriptions in other datasets as well.

Since in both Gnome and Eclipse, stack traces are stored in the description by the user, not necessarily all of the bug reports have stack traces. Whereas in Eclipse, 10% of bug reports have stack traces in the description, because in 2015 Eclipse established an automatic stack trace collection system, we expected more bug reports to have stack traces from 2015 onwards. These stack traces are not made publicly available yet. For the Gnome dataset, 32% of bug reports had stack traces in their description, which is a good ratio.

7.4.2. Threats to Internal Validity

In our approach, the misclassification cost is calculated using our own proposed heuristic based on Equation (15). The outcome of the approach using that heuristic is promising. However, the parameter

could be adjusted using an exhaustive domain search or training machine learning methods. Using a more optimal parameter could further enhance the product and component prediction capability of our approach.

Another possible threat may be concerning the use of regular expressions to extract stack traces from bug report descriptions. Our regular expression may have missed some stack traces. The missed stack traces could have slightly altered the accuracy of our approach.

7.4.3. Threats to Construct Validity

The construct validity shows how the used evaluation measures could reflect the performance of our predictive model. In this study, we used precision, recall and F-measure. These measures are widely used in other studies to assess the accuracy of machine learning models (e.g., [SHH16, XLSW16, LPG02, SM12]).

7.5. Conclusions and Future Work

In this chapter, we have proposed a new approach to predict faulty product and component fields of bug reports. Our approach leverages stack traces and categorical features instead of bug report descriptions. In our approach, we have used a linear combination of stack traces and categorical features similarity to predict product and component fields. We have also used the cost-sensitive K-nearest neighbour method to overcome the unbalanced dataset distribution problem and predict faulty products and components.

To assess the accuracy of our approach, we used two well-studied and publicly available datasets (Eclipse and Gnome). We have assessed the performance of our approach using 221,038 bug reports from Gnome and Eclipse. Experiment results showed that our approach could predict the faulty product and component fields by an average F-measure of 65%. We have also shown that using stack traces and categorical features as a more formal source of information could improve the bug report faulty product and component field prediction accuracy. We showed that our approach using stack traces and

categorical features outperforms the one that uses bug report descriptions to predict faulty product and component fields of bug reports by 35% on average.

Our approach can be used effectively to predict faulty product and component fields of bug reports to eliminate the overhead of the wrong assignment of the bug reports to the development teams. The proposed approach could significantly reduce the software maintenance cost by efficient localization of software bugs.

As future work, we plan to evaluate our approach using additional datasets. We also plan to use a training model to obtain the optimized misclassification cost for each product and component. Furthermore, we plan to use more advanced machine learning techniques.

Chapter 8

Conclusion and future work

Handling bug reports after the release of a software system is an end-users and time-consuming part of software maintenance. Bug tracking systems are widely used to capture software bugs from the end-users. These bug reports need to be handled by the software company. Bug tracking systems require end-users to provide technical information when submitting software bug reports. Bug triagers mainly rely on the information provided by the users to route the bug reports to the development teams. Bug report handling process suffers from many shortcomings. In practice, the end-users are usually unfamiliar with the structure and technical aspects of the software system. The information they tend to provide is limited and often error-prone. Since bug triagers mainly rely on the information provided by users, incorrect data misleads them and consequently delays the resolution of bugs. There is a need for tools that can alleviate the necessity of providing information from users.

There exist studies for automatically predicting the information needed for triaging bug reports. However, these techniques mainly rely on the description of bug reports, which are written by end-users and hence tend to be error-prone. In this thesis, we studied the effectiveness of using stack traces. Stack traces are automatically generated by the software systems when a crash happens and do not require end-user to provide any information about the bug (reducing the human error element).

We showed that our proposed approaches in Chapters 5, 6 and 7 can be used to automate the main steps of a bug triaging process with an acceptable level of accuracy. We believe that an automatic bug triaging system could significantly improve the accuracy of bug assignment and bug fixing process, which can result in cost savings. Furthermore, the proposed methods for automating the bug triaging process in this thesis are designed to be easily deployable.

In this chapter, we provide a summary of our findings and introduce future research directions.

8.1. Thesis Findings

In this section, we provide a summary of our findings.

We compared the time to detect duplicate of an incoming bug report in the presence and absence of stack traces and found that the duplicate detection time for bug reports, which have stack traces, is statistically significantly different from duplicate detection time of bug reports without stack traces. We also found that duplicate bug report detection time in the presence of stack traces is lower compared to bug reports without stack traces.

We studied the relation between the presence of stack traces in bug reports and the severity of the bugs. We found that there is a strong statistical association between the existence of stack traces and the severity of bug reports.

We found that there is a significant statistical association between the reassignment of bug report faulty product and component fields and the existence of stack traces in the bug reports. We found that although faulty product and component fields of bug reports with stack traces are reassigned more than bug reports without stack traces, bug reports for which faulty product or component fields are reassigned are fixed sooner if they have stack traces.

When a bug report is received in the bug tracking system, bug triagers assess if it is a duplicate bug report. We created groups of duplicate bug reports and found that in Eclipse 68% of duplicate bug reports groups contain only two bug reports.

We showed that since open source software systems such as Eclipse grow quickly, using functions in stack traces to detect duplicate bug reports leads to the curse of dimensionality problem. We devised DURFEX, a duplicate detection approach that overcomes the curse of dimensionality problem by leveraging trace abstraction techniques. We showed that DURFEX not only eliminates the curse of dimensionality problem

by reducing the feature set almost 70% but also detects duplicate bug reports more accurately compared to approaches that are based on sequences of function calls.

If a bug report is not duplicate of a previously triaged bug report, triagers need to assess the severity of the bug. We devised an approach for predicting the severity of bugs using stack traces. We showed that stack traces could be used to predict the severity of bugs. We also showed that stack traces could predict the severity of bugs with a statistically higher accuracy compared to bug report description provided by the end-users when submitting bug reports. We also showed that severity prediction accuracy is higher using stack traces compared to the approach which predicts severity labels randomly.

We found that, in addition to stack traces, using categorical features such as faulty product, faulty component and operating system could further increase bug severity prediction accuracy. We devised a new approach, which predicts the severity of bugs using stack traces and categorical features. We showed that severity prediction accuracy is statistically significantly higher when using stack traces and categorical features compared to using stack traces only. We also showed the approach which uses stack traces and categorical features predict severity with higher accuracy compared to the approach, which predicts severity labels randomly.

The last step in the bug triaging process is routing bug report to the development team who is responsible for the faulty product and component. We designed a new approach that predicts faulty product and component fields of bug reports based on stack traces and categorical features. We showed that the prediction accuracy is higher using the devised approach compared to using the bug report description provided by the end-user. We also show that the faulty product and component prediction accuracy is higher using the proposed approach compared to the approach which predicts faulty product and component fields of bug reports randomly [SHTH19, SNTH18].

8.2. Future Work

8.2.1. Limitations

One of the limitations of this thesis is the availability of stack traces. At the present time, stack traces are manually copied to the description of bug reports in many bug tracking systems. This may explain why only 10% of Eclipse bug reports contain stack traces. Eclipse recently started an automatic system for collecting stack traces when a crash occurs. However, stack traces are not made available publicly yet. In Gnome, similar to Eclipse, stack traces are copied to the description of bug reports. However, the number of stack traces in Gnome is higher compared to Eclipse. In the Gnome dataset used for severity prediction, 45% of bug reports had stack traces and, in the dataset, used for prediction of product and component fields, 32% of bug reports had stack traces. This limitation can be further resolved once software systems start providing automatic stack trace collection systems, and when bug tracking systems made stack traces publicly available.

Another limitation is the misclassification heuristic used for the cost-sensitive K nearest neighbour method. The misclassification cost can be adjusted more precisely by an exhaustive domain search. It also could be adjusted using machine learning methods. In our studies, we set the cost of the misclassification of each class label to be reciprocal to the number of existing instances of that class divided by the number of instances of the majority class. Using a more optimal parameter could further enhance the severity, product and component field prediction capability of our approaches.

One other limitation is the way we implemented the approach for extracting words from bug report descriptions. We simply tokenized each bug report description and used the extracted words to form feature vectors. We did not resort to any natural language processing method. The use of a powerful natural language processing method may result in better performance of an approach that uses bug report descriptions.

The regular expressions used to extract stack traces from bug report descriptions may miss some complex cases. We used the regular expression provided by Lerch et al. [LM13] to extract stack traces from Eclipse bug repository, but for Gnome since to the best of our knowledge, there is no publicly available tool for extracting stack traces, so we defined our own regular expression. To mitigate this threat, we manually verified hundreds of extracted traces from the Gnome dataset using our regular expression. Based on our evaluation, our regular expression extracts stack traces from Gnome bug reports with an accuracy of 95%.

In this thesis, we relied on the labels provided by the triagers in the bug tracking systems. Although triagers are usually knowledgeable of the system, errors may occur, which can affect the accuracy of the proposed approaches.

In this thesis, we focused on bugs that cause software systems to crash. However, some bugs do not lead to crashes. If a bug does not lead to a crash, we cannot generate a stack trace, which limits the applicability of the proposed approaches.

Similar to most of the classifiers, K-nearest neighbour is also sensitive to noise. In our experiments, we used the labels provided by the triagers. However, there could be some noise in the data, which may render K-nearest neighbour approach sensitive and therefore impacts its performance.

8.2.2. Future Research Opportunities

We should aim to assess the effectiveness of stack traces in other bug report processing tasks. We can apply the proposed methods to predict bug reports that may be re-opened. Re-opened bug reports are bug reports, which are revisited because the fix was not satisfactory. Re-opened bug reports also have longer processing time [SIKIO+10] than the not re-opened bug reports. In the literature, bug report descriptions are used for predicting re-opened bug reports [SIKIO+10]. An interesting research direction is to use stack traces to see if better techniques can be developed.

In our studies, we leveraged stack traces to predict different bug report fields. We showed that stack traces can help predict bug report fields more accurately than bug report descriptions. One interesting future research direction is to use other types of information such as domain expertise. In addition, we need to study how the combination of various sources of data (traces, descriptions, domain knowledge, etc.) can be combined and which context they can be used either separately or by combining them.

We used stack traces for predicting severity and product and component fields of bug reports. We use distinct method names in all the stack traces as features. Software systems grow quickly, and as a result, more features will be introduced. Excessive increase in the number of features leads to the curse of dimensionality problem and limits the practical applicability of the proposed approaches. In Chapter 5, we introduced DURFEX [SHL17], which leverages trace abstraction concepts to reduce the number of features used for detecting duplicate bug reports. One future area of research can be to assess the accuracy of predicting severity and faulty product and component fields of bug reports using trace abstraction techniques. In addition, we need to investigate the use of other trace abstraction techniques such as the ones used in program comprehension [PH11, PAH10, HL04, HL06, HLF05].

Another future avenue of improvement is the use of more advanced machine learning methods. We used a linear model and cost-sensitive K-nearest neighbour method for our studies. There are future opportunities for using more advanced machine learning methods such as deep learning methods to further improve the prediction accuracy of the proposed approaches.

References

[AADPKG08] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y. Gueheneuc, “Is it a bug or an enhancement? A text-based approach to classify change requests”, In *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, CASCON '08, pages 23:304-23:318, New York, NY, USA, 2008. ACM.

[AHS13] A. Alipour, A. Hindle, and E. Stroulia, “A contextual approach towards more accurate duplicate bug report detection”, In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 183-192, Piscataway, NJ, USA, 2013.

[AO08] P. Amman and J. Offutt, Introduction to Software Testing. Cambridge university press, New York, 2008

[BINF12] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos, “Graph-based analysis and prediction for software evolution”, In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. Piscataway, NJ, USA, 419-429.

[BN10] P. Bhattacharya and I. Neamtiu, "Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging", In *Proceedings of 2010 IEEE International Conference on Software Maintenance*, Timisoara, 2010, pp. 1-10.

[BJSWPZ08] N. Bettenburg, S. Just, A. Schroter, C. Weiss, R. Premraj, and T. Zimmermann, “What makes a good bug report?”, In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT'08/FSE-16, pages 308-318, New York, NY, USA, 2008.

[BMLSM+05] M. Brodie, S. Ma, G. Lohman, T. syeda-mahmood, L. Mignet, N. Modani, M. Wilding, J. Champlin, and P. Sohn, “Quickly finding known software problems via automated symptom matching”, In *Proceedings of the Second International Conference on Autonomic Computing*, pages 101-110, June 2005.

[BPSZ09] S. Breu, R. Premraj, J. Sillito, and T. Zimmermann, “Frequently asked questions in bug reports,” *University of Calgary, Technical Report*, 2009.

[BPZK08a] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, “Duplicate bug reports considered harmful… really?”, In *Proceedings of the 2008 IEEE International Conference on Software Maintenance*, pages 337-345, Sept 2008.

[BPZK08b] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, “Extracting structural information from bug reports”, In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, MSR '08, pages 27-30, New York, NY, USA, 2008.

[CSVP17] M. Castelluccio, C. Sansone, L. Verdoliva, and G. Poggi, “Automatically analyzing groups of crashes for finding correlations,” *In Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, pp. 717–726, 2017.

[DMJ11] J. Davidson, N. Mohan, and C. Jensen, “Coping with duplicate bug reports in free/open-source software projects”, In *Proceedings of 2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 101-108, Sept 2011.

[DWZZN12] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel, "Rebucket: A method for clustering duplicate crash reports based on call stack similarity", In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1084-1093, Piscataway, NJ, USA, 2012.

[EIHH16] N. Ebrahimi, M. S. Islam, A. Hamou-Lhadj, and M. Hamdaqa, "An Effective Method for Detecting Duplicate Crash Reports Using Crash Traces and Hidden Markov Models," In *Proceedings of the IBM 26th Annual International Conference on Computer Science and Software Engineering (CASCON)*, Toronto, ON, Canada, pp. 75-84, 2016.

[ETIHK19] N. Ebrahimi, A. Trabeslsi, M. S. Islam, A. Hamou-Lhadj, and K. Khanmohammadi, "An HMM-Based Approach for Automatic Detection and Classification of Duplicate Bug Reports," *Elsevier Journal of Information and Software Technology* (IST), 2019.

[EH15] N. Ebrahimi, and A. Hamou-Lhadj, "CrashAutomata: An Approach for the Detection of Duplicate Crash Reports Based on Generalizable Automata", *In Proceedings of the IBM Centers for Advanced Studies Conference* (CASCON), pp. 201-210, 2015.

[FAA16] A. Florea, J. Anvik, and R. Andonie, "Parallel Implementation of a Bug Report Assignment Recommender Using Deep Learning", *In Proceedings of the International Conference on Artificial Neural Networks (ICANN)*, 2016, pp. pp 64-71

[FFHL05] J. Feller, B. Fitzgerald, S. Hissam, and K. Lakhani, "Perspectives on Free and Open Source Software", *The MIT Press*, 2005.

[GK05] R. Grissom and J. Kim, "Effect sizes for research: A Broad Practical Approach", *Journal of Developmental & Behavioral Pediatrics, Lawrence Erlbaum Associates, 2006*

[GPG10] E. Giger, M. Pinzger, and H. Gall, "Predicting the fix time of bugs", *In Proceedings of the second International Workshop on Recommendation Systems for Software Engineering*, 2010, pp. 52–56.

[GS14] H. Valdivia Garcia, and E. Shihab, "Characterizing and predicting blocking bugs in open source projects", In *Proceedings of the 11th Working Conference on Mining Software Repositories,* Pages 72-81, 2014, Hyderabad, India, 2014.

[GDZX12] J. Gou, L. Du, Y. Zhang, and T. Xiong, "A New Distance-weighted k-nearest Neighbour Classifier" *Journal of Information & Computational Science*, pages 1429–1436, 2012.

[GZNM11] P. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, "not my bug! and other reasons for software bug report reassignments", *In Proceedings of the Conference on Computer Supported Cooperative Work* , 2011, pp. 395–404.

[HL04] A. Hamou-Lhadj, and T. Lethbridge, "Reasoning About the Concept of Utilities," In *Proceedings of ECOOP International Workshop on Practical Problems of Programming in the Large*, Oslo, Norway, Lecture Notes in Computer Science (LNCS), Vol 3344, Springer-Verlag, pp. 10-22, 2004.

[HL06] A. Hamou-Lhadj, and T. Lethbridge, "Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System", In *Proceedings of the IEEE 14th International Conference on Program Comprehension*, IEEE CS, pp. 181-190, 2006.

[HLF05] A. Hamou-Lhadj, and T. Lethbridge, and L. Fu,"SEAT: A Usable Trace Analysis Tool," In *Proceedings of the International Workshop on Program Comprehension (IWPC), renamed the International Conference on Program Comprehension (ICPC),* pp. 157-160, 2005.

[JW08] N. Jalbert and W. Weimer, "Automated duplicate detection for bug tracking systems". In *Proceedings of 2008. IEEE International Conference of Dependable Systems and Networks,* 2008*.* pages 52-61, June 2008.

[K12] M. Krikke, "Investigating the usefulness of stack traces in bug triaging", *Master Thesis, Delft University of Technology*

[KZN11] S. Kim, T. Zimmermann, and N. Nagappan, "Crash graphs: An aggregated view of multiple crashes to improve crash triage", In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks, DSN '11*, pages 486-493, Washington, DC, USA, 2011

[LD13] A. Lamkanfi and S. Demeyer, "Predicting reassignments of bug reports an exploratory investigation", *In Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR)*, 2013, pp. 327–330.

[LDGG10] A. Lamkan, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug", In *Proceedings of 7th IEEE Working Conference on Mining Software Repositories (MSR),* pages 1-10, May 2010.

[LDSV11] A. Lamkanfi, S. Demeyer, Q. David Soetens, and T. Verdonck, "Comparing mining algorithms for predicting the severity of a reported bug", *In proceedings of 15th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 249-258, March 2011.

[LK12] W. Le, and D. Krutz, "How to Group Crashes Effectively: Comparing Manually and Automatically Grouped Crash Dumps", *Technical Report Rochester institute of technology*, 2012

[LM13] J. Lerch, and M. Mezini, "Finding duplicates of your yet unwritten bug report". *In proceedings of 17th European Conference on Software Maintenance and Reengineering (CSMR), 2013*, pages 69-78, March 2013.

[LPG02] G. A. Di Lucca, M. Di Penta, and S. Gradara, "An approach to classify software maintenance requests", *In Proceedings of the International Conference on Software Maintenance (ICSM), 2002*, Montreal, Quebec, Canada, 2002,pp.93-102.

[MHNSL15] A. Maiga, A. Hamou-Lhadj, M. Nayrolles, K.K. Sabor and A. Larsson, "An empirical study on the handling of crash reports in a large software company: An experience report", *In Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME), 2015*

[MGLMM07] N. Modani, R. Gupta, G. Lohman, T. Syeda-Mahmood, and L. Mignet, "Automatically identifying known software problems", *In proceedings of IEEE 23rd International Conference on Data Engineering Workshop*, pages 433-441, April 2007.

[MM08] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports", *In proceedings of 2008 IEEE International Conference on Software Maintenance*, pages 346-355, 2008.

[MRL11] G. Macbeth, E. Razumiejczyk, and R. Ledesma, "Cliff's Delta Calculator: A non-parametric effect size program for two groups of observations", Universitas Psychologica. 10. 545-555, (2011).

[MRS08] C. D. Manning, P. Raghavan, and H. Schütze, Introduction to Information Retrieval. *Cambridge University Press*, NY, USA, 2008.

[NHTL15] M. Nayrolles, A. Hamou-Lhadj, S. Tahar and Alf Larsson, "JCHARMING: A Bug Reproduction Approach Using Crash Traces and Directed Model Checking," In *proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (a merger of WCRE and CSMR) (SANER'15),* 2015.

[NHTL16] M. Nayrolles, A. Hamou-Lhadj, S. Tahar, and Alf Larsson, "A Bug Reproduction Approach Based on Directed Model Checking and Crash Traces," *Wiley Journal of Software: Evolution and Process (JSPE)*, 2016.

[N2] M. Newman, "Software errors cost us economy $59.5 billion annually," *NIST Assesses Technical Needs of Industry to Improve Software Testing*, 2002.

[NNNLS12] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun, "Duplicate bug report detection with a combination of information retrieval and topic modelling", In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012*, pages 70-79, New York, NY, USA, 2012. ACM.

[PAH10] H. Pirzadeh, A. Agarwal, A. Hamou-Lhadj, "An Approach for Detecting Execution Phases of a System for the Purpose of Program Comprehension," In *Proceedings. of the 8th International Conference on Software Engineering Research, Management & Applications* (SERA 2010), pp. 207 - 214, 2010.

[PF13] F. Provost, and T. Fawcett**, "**Data Science for Business: What You Need to Know about Data Mining and Data-Analytic Thinking", *O'Reilly,* 2013, New York

[PH11] H. Pirzadeh, and A. Hamou-Lhadj, "A Novel Approach Based on Gestalt Psychology for Abstracting the Content of Large Execution Traces for Program Comprehension", In *Proceedings of the 16th IEEE International Conference on Engineering of Complex Computer Systems* (ICECCS '11), 2011

[QWZZ13] Z. Qin, A. T. Wang, C. Zhang, and S. Zhang, "Cost-Sensitive Classification with k-Nearest Neighbours", In *Proceedings of Knowledge Science, Engineering and Management: 6th International Conference,* pages 112-131, Dalian, China, August 2013.

[RAN07] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing", In *Proceedings of 29th International Conference on Software Engineering, 2007. ICSE 2007*, pages 499-510, May 2007.

[S12] A. Sureka, "Learning to classify bug reports into components", *In Proceedings Of the 50th International Conference on Objects, Models, Components, Patterns: 50th International Conference (TOOLS)*, 2012, pp. 288–303.

[SBP10] A. Schroter, N. Bettenburg, and R. Premraj, "Do stack traces help developers fix bugs?", In *Proceedings of 7th IEEE Working Conference on the Mining Software Repositories (MSR), 2010*, pages 118-121, May 2010

[SHH16] K. K. Sabor, M. Hamdaqa, and A. Hamou-Lhadj, "Automatic prediction of the severity of bugs using stack traces", *In Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering (CASCON)*, 2016, pp. 96–105.

[SHH19] K. K. Sabor, M. Hamdaqa, and A. Hamou-Lhadj, "Automatic prediction of the severity of bugs using stack traces and categorical features", *In Elsevier Journal on Information and Software Technology, 2019.*

[SHL17] K. K. Sabor, A. Hamou-Lhadj and A. Larsson, "DURFEX: A Feature Extraction Technique for Efficient Detection of Duplicate Bug Reports, *In proceedings of International Conference on Software Quality, Reliability and Security (QRS)*, Prague, 2017, pp. 240-250.

[SHTH19] K. K. Sabor, A. Hamou-Lhadj, A. Trabelsi, and J. Hassine. "Predicting bug report fields using stack traces and categorical attributes", In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering (CASCON '19)*. IBM Corp., USA, 224–233.

[SM12] K. Somasundaram and G. Murphy, "Automatic Categorization of Bug Reports Using Latent Dirichlet Allocation", *In the Proceedings of the 5th India Software Engineering Conference (ISEC)*, 2012, pp. 125-130

[SNTH18] K. K. Sabor, M. Nayrolles, A. Trabelsi and A. Hamou-Lhadj, "An Approach for Predicting Bug Report Fields Using a Neural Network Learning Model," *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Memphis, TN, 2018, pp. 232-236.

[SSG15] G. Sharma, S. Sharma, and S. Gujral, "A Novel Way of Assessing Software Bug Severity Using Dictionary of Critical Terms", *In Proceedings of the 4th International Conference on Eco-friendly Computing and Communication Systems,* Volume 70, 2015, Pages 632-639, ISSN 1877-0509, Procedia Computer Science,

[SIKIO+10] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K. Matsumoto, "Predicting reopened bugs: A case study on the Eclipse project", *In Proceedings of the 17th Working Conference on Reverse Engineering (WCRE),* 2010, pp. 249–258.

[SJ10] A. Sureka and P. Jalote, "Detecting duplicate bug report using character n-gram-based features", *In Proceedings of the 17th Asia Pacific Software Engineering Conference (APSEC), 2010*, pages 366-374, Nov 2010.

[SLKJ11] C. Sun, D. Lo, S. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports", In *proceedings of 26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 253-262, Nov 2011.

[SLWJK10] C. Sun, D. Lo, X. Wang, J. Jiang, and S. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval", In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 45-54, New York, NY, USA, 2010. ACM.

[TLS12] Y. Tian, D. Lo, and C. Sun, "Information retrieval based nearest neighbour classification for fine-grained bug severity prediction", In *Proceedings of the 2012 19th Working Conference on Reverse Engineering (WCRE)*, pages 215-224, Oct 2012.

[WKZ13] S. Wang, F. Khomh, and Y. Zou, "Improving bug localization using correlations in crash reports", *In Proceedings of 10th IEEE Working Conference Mining Software Repositories (MSR),* pages 247-256, May 2013.

[WZLLW12] D. Wang, H. Zhang, R. Liu, M. Lin, and W. Wu, "Predicting Bugs' Components via Mining Bug Reports," *Journal of Software*, 7(5), 2012, pp. 1149-1154.

[XLWSZ14] X. Xia, D. Lo, M. Wen, E. Shihab, and B. Zhou, "An empirical study of bug report field reassignment," *In proceedings of IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, Feb 2014, pp. 174–183

[XLSW16] X. Xia, D. Lo, E. Shihab, and X. Wang, "Automated bug report field reassignment and refinement prediction," *In IEEE Transactions on Reliability*, 65(3), 2016, pp. 1094–1113

[XZM13] J. Xie, M. Zhou, and A. Mockus, "Impact of Triage: A Study of Mozilla and Gnome," *In proceedings. of the International Symposium on Empirical Software Engineering and Measurement*, 2013, pp. 247-250

[YCKY14] C. Yang, K. Chen, W. Kao, and C. Yang, “Improving severity prediction on software bug reports using quality indicators”, *In Proceedings of the 5th IEEE International Conference on Software Engineering and Service Science* (ICSESS’14), pages 216–219, 2014

[YHKC12] C. Yang, C. Hou, W. Kao, and I. Chen, “An empirical study on improving severity prediction of defect reports using feature selection”, In *Proceedings of the 2012 19th Asia-Pacific Software Engineering Conference – Volume 01 (APSEC ’12)*, pages 240–249, 2012.

[YZL14] G. Yang, T. Zhang, and B. Lee, “Towards semi-automatic bug triage and severity prediction based on topic model and multi feature of bug reports”, *In Proceedings of the 38th Annual Computer Software and Applications Conference (COMPSAC’14)*, pages 97–106, 2014.

[ZM03] J. Zhang, and I. Mani, “KNN Approach to Unbalanced Data Distributions: A case study involving Information Extraction”, In *proceedings of the 20’th international conference on machine learning,* 2003

[ZCYLL16] T. Zhang, J. Chen, G. Yang, B. Lee, and X. Luo, ”Towards more accurate severity prediction and fixer recommendation of software bugs”, In *Journal of Systems and Software*, Volume 117, 2016, Pages 166-184

[ZYLC15] T. Zhang, G. Yang, B. Lee, and A. T. S. Chan, “Predicting severity of bug report by mining bug repository with concept profile”, In *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC’15)*, pages 1553–1558, 2015

[ZWW15] W. Zhang, S. Wang, and Q. Wang, “KSAP: An approach to bug report assignment using KNN search and heterogeneous proximity”, *Journal of Information and Software Technology*, Volume 70, 2015, pp. 68–84.