MINING SOFTWARE REPOSITORIES FOR AUTOMATIC SOFTWARE BUG MANAGEMENT

FROM BUG TRIAGING TO PATCH BACKPORTING

YUAN TIAN

SINGAPORE MANAGEMENT UNIVERSITY

2017

Mining Software Repositories for Automatic Software Bug Management

From Bug Triaging to Patch Backporting

by

Yuan Tian

Submitted to School of Information Systems in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Information Systems

Dissertation Committee:

David Lo (Chair)
Associate Professor of Information Systems
Singapore Management University

Jing Jiang
Associate Professor of Information Systems
Singapore Management University

Lingxiao Jiang
Assistant Professor of Information Systems
Singapore Management University

Julia Lawall
Director of Research
Inria - Laboratoire d'Informatique de Paris 6

Singapore Management University
2017

Mining Bug Repositories for Automatic Software Bug Management

From Bug Triaging to Patch Backporting

Yuan Tian

Abstract

Software systems are often released with bugs due to system complexity and inadequate testing. Bug resolving process plays an important role in development and evolution of software systems because developers could collect a considerable number of bugs from users and testers daily. For instance, during September 2015, the Eclipse project received approximately 2,500 bug reports, averaging 80 new reports each day. To help developers effectively address and manage bugs, bug tracking systems such as Bugzilla and JIRA are adopted to manage the life cycle of a bug through bug report. Since most of the information related to bugs are stored in software repositories, e.g., bug tracking systems, version control repositories, mailing list archives, etc. These repositories contain a wealth of valuable information, which could be mined to automate bug management process and thus save developers time and effort.

In this thesis, I target the automation of three bug management tasks, i.e., bug prioritization, bug assignment, and stable related patch identification.

Bug prioritization is important for developers to ensure that important reports are prioritized and fixed first. For automated bug prioritization, we propose an approach that recommends a priority level based on information available in bug reports by considering multiple factors, including temporal, textual, author, related-report, severity, and product, that potentially affect the priority level of a bug report. After being prioritized, each reported bug must be assigned to an appropriate developer/team for handling the bug. This bug assignment process is important, because assigning a bug report to the incorrect developer or team can increase the overall

time required to fix the bug, and thus increase project maintenance cost. Moreover, this process is time consuming and non-trivial since good comprehension of bug report, source code, and team members is needed. To automate bug assignment process, we propose a unified model based on learning to rank technique. The unified model naturally combines location-based information and activity-based information extracted from historical bug reports and source code for more accurate recommendation. After developers have fixed their bugs, they will submit patches that could resolve the bugs to bug tracking systems. The submitted patches will be reviewed and verified by other developers to make sure their correctness. In the last stage of bug management process, verified patches will be applied on the software code. In this stage, many software systems prefer to maintain multiple versions of software systems. For instance, developers of the Linux kernel release new versions, including bug fixes and new features, frequently, while maintaining some older "longterm" versions, which are stable, reliable, and secure execution environment to users. The maintaining of longterm versions raises the problem of how to identify patches that are submitted to the current version but should be backported to the longterm versions as well. To help developer find patches that should be moved to the longterm stable versions, we present two approaches that could automatically identify bug fixing patches based on the changes and commit messages recorded in code repositories. One approach is based on hand-crafted features and two machine learning techniques, i.e., LPU (Learning from Positive and Unlabeled Examples) and SVM (Support Vector Machine). The other approach is based on a convolutional neural network (CNN), which automatically learns features from patches.

Table of Contents

List of Figures

List of Tables

Acknowledgment

I would like to express my gratitude to my supervisor, collaborators, dissertation committee members, friends, university staffs, and family members, for their kindly help in many ways smoothing the progress of my PhD.

First and foremost, I thank my supervisor, Prof. David Lo, for his consistently support and encouragement during my PhD training. Next, I thank my dissertation committee members: Prof. Jing Jiang, Prof. Lingxiao Jiang, and Julia Lawall, for taking time reviewing my thesis, providing valuable suggestions, and attending my defense. Julia is especially kind for flying all the way to Singapore from France to attend my defense. I thank my collaborators from worldwide institutions, for their help in my research. They are Julia Lawall from Inria, Prof. Ahmed E. Hassen from Queen's University, Xin Xia from Zhejiang University, Prof. Meiyappan Nagappan from University of Waterloo, Prof. Claire Le Goues from Carnegie Mellon University, and Dr. Chengnian Sun, who is now working at Google. I acknowledge the friendship and support from my group members in the SOAR (SOftware Analytics Research) lab of SMU and my friend Ke Xu. I am also grateful to the following university staffs: Seow Pei Huan and Ong Chew Hong, for their unfailing support and assistance, especially in managing the timeline of my graduation process.

Last but not the least, I am grateful to my parents, Kangsheng Tian and Xiaoyan Wang, for their generous care and believes in me. Thanks my husband, Shuang Xia, who is working hard for our small family. He makes my life brighter and much easier :-)

Chapter 1

Introduction and Overview

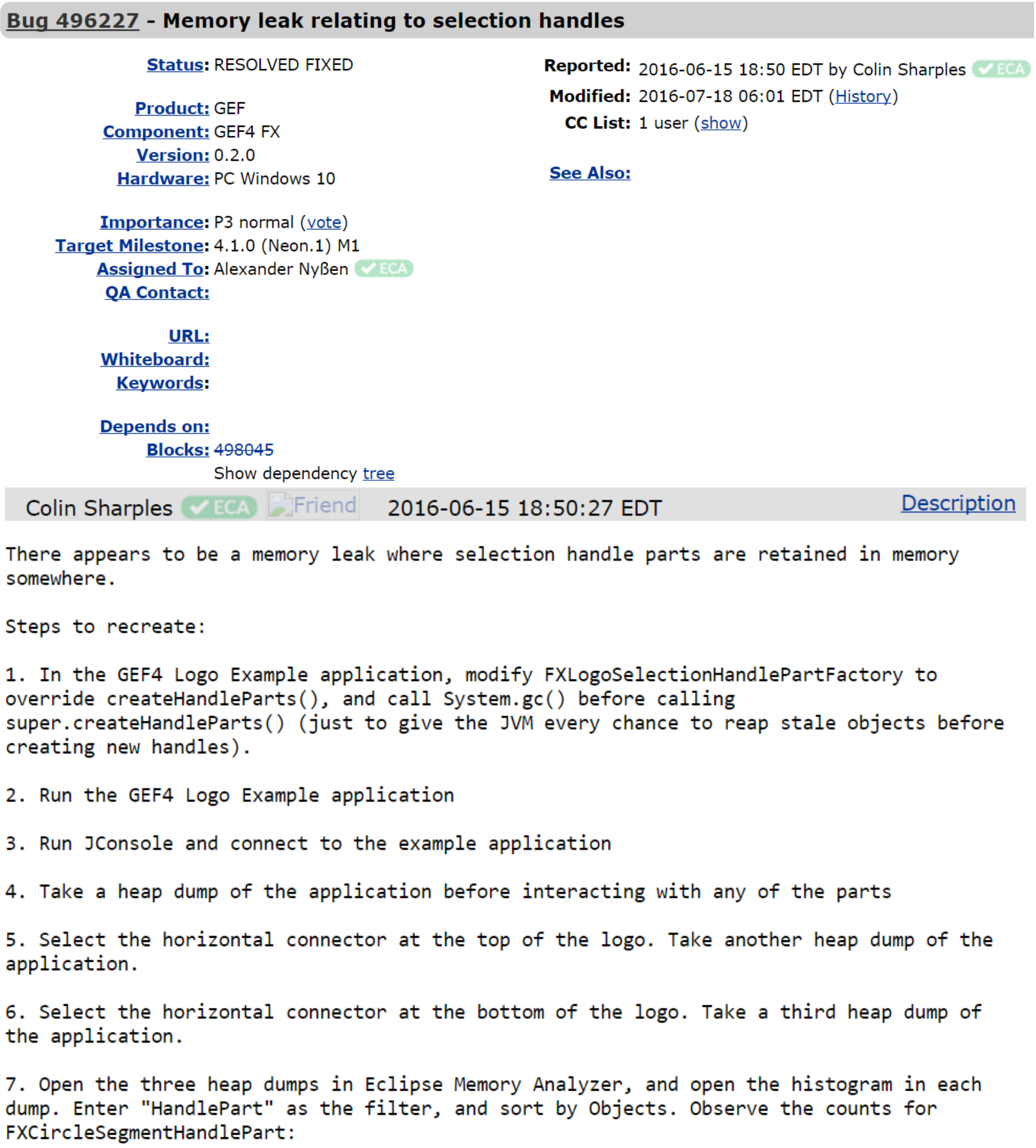
1.1 The Life Cycle of a Bug

Due to system complexity and inadequate testing, many software systems are often released with defects. To address these defects and improve the next releases, developers need to get feedback on defects that are present in released systems. Thus, they often allow users/testers to report defects using bug tracking systems such as Bugzilla,[1] JIRA,[2] or other proprietary systems. Bug tracking is a standard practice in both open source software development and closed source software development. Figure 1.1 presents a sample bug report from the Eclipse project that is stored in Bugzilla. It shows that bug tracking system has provided many fields to help software maintainers to manage a bug, e.g., the status of the bug, the reporter of the bug, the product/component impacted by the bug, etc. A bug report also contains textual information such as a summary of the bug, steps to reproduce the bug, etc. The values of some fields are provided by the bug reporter when the bug is submitted to the system, while the values of other fields are provided or updated by software maintainers after the bug is reported.

Every defect or bug that has been discovered goes through a process before it is

[1] https://www.bugzilla.org/

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



Bug 496227 - Memory leak relating to selection handles

Status: RESOLVED FIXED

Product: GEF
Component: GEF4 FX
Version: 0.2.0
Hardware: PC Windows 10

Importance: P3 normal (vote)
Target Milestone: 4.1.0 (Neon.1) M1
Assigned To: Alexander Nyßen ✓ECA
QA Contact:

URL:
Whiteboard:
Keywords:

Depends on:
Blocks: ~~498045~~
Show dependency tree

Reported: 2016-06-15 18:50 EDT by Colin Sharples ✓ECA
Modified: 2016-07-18 06:01 EDT (History)
CC List: 1 user (show)

See Also:

Colin Sharples ✓ECA Friend 2016-06-15 18:50:27 EDT Description

There appears to be a memory leak where selection handle parts are retained in memory somewhere.

Steps to recreate:

1. In the GEF4 Logo Example application, modify FXLogoSelectionHandlePartFactory to override createHandleParts(), and call System.gc() before calling super.createHandleParts() (just to give the JVM every chance to reap stale objects before creating new handles).

2. Run the GEF4 Logo Example application

3. Run JConsole and connect to the example application

4. Take a heap dump of the application before interacting with any of the parts

5. Select the horizontal connector at the top of the logo. Take another heap dump of the application.

6. Select the horizontal connector at the bottom of the logo. Take a third heap dump of the application.

7. Open the three heap dumps in Eclipse Memory Analyzer, and open the histogram in each dump. Enter "HandlePart" as the filter, and sort by Objects. Observe the counts for FXCircleSegmentHandlePart:

Figure 1.1: A Sample Bug Report from Eclipse Project

resolved. Different organizations might have slightly different strategies to manage bugs, but the overall life cycle of a bug is usually similar. In this thesis, we segment the whole bug management process into four stages, i.e., bug detection & reporting, bug triaging, debugging & bug fixing, and patch verification & backporting. Each of these stages is described below.

Stage 1: Bug Detection & Reporting. Initially, a bug is detected or encountered by bug detection tools, testers or users. In order to detect bugs at an early stage of software development, multiple bug detection techniques have been introduced [8, 30, 70, 77]. After a bug is detected, a developer/user (i.e., bug reporter) can file a report in the bug tracking system. Inside the bug report, the developer should describe what is the bug and how to reproduce the bug, and provide values of fields such as product/component impacted by the bug, version of the software, severity of the bug, etc. These information will help software maintainers to resolve the bug.

Stage 2: Bug Triaging. Bug tracking systems receive a large number of bug reports daily. Each reported bug must be scanned by developers to determine if it describes a meaningful problem, and if it does, it must be prioritized and assigned to an appropriate developer for further handling. Such a procedure is called bug triaging. Bug triaging process contains many sub-tasks, some of which are introduced below.

- **Duplicate bug report detection** identifies whether a reported bug has already been reported. It helps developers to filter out duplicate bug reports thus saves developers' time in fixing redundant bugs.

- **Bug prioritization** ranks new bug reports and assigns a priority level to each bug to prioritize which bugs should be given attention first. Prioritizing bugs is needed due to limited human resources. It is a manual process and is time consuming. Bug triagers need to read the information provided by bug reporters in the new bug reports, compare them with existing reports, and choose the appropriate priority levels.

- **Bug assignment** decides who should fix a reported bug. Appropriate bug assignment is important because a bug could be fixed faster if it is assigned to the right person. Usually bug triagers will consider the description in the reported bug and find the developer who has taken charge of related source code files or/and who has enough expertise to fix the bug.

Stage 3: Debugging & Bug Fixing. After the developer has received a bug, he/she will start to diagnose the cause of the bug based on the information available in the bug report. For instance, the developer might try to reproduce the bug following the description in the report and then locate the buggy code that cause the bug. Such procedure of finding the location of buggy code for a reported bug based on its report is called *bug report localization*. Once the developer figures out where is the buggy code, he/she will create a patch that could be applied on the source code to fix the bug. The patch will be submitted to the bug tracking system for further verification.

Stage 4: Patch Verification & Backporting. Given a patch created by a developer to fix a bug, some other developers need to verify whether this patch could be merged into the new version of the software. Such a patch verification procedure might iterate several times between the patch creater and the patch reviewer until the patch is ready for merging. After a patch is verified, the reported bug is resolved and the patch will be merged to the particular version of the software. For some software, bug fixing patches should be back-ported to the older version of the same software to improve the usability of older versions. Such process is called *patch backporting*. In the patch backporting process, a maintainer will forward the patch to the maintainers of the longterm versions, if the patch satisfies various guidelines, such as fixing a real bug, and making only a small number of changes to the code.

In this thesis, we focus on two of the four mentioned stages in the management of bugs, i.e., bug triaging and patch verification & back-porting. Note that for patch verification & backporting step, we only target one task, i.e., bug fixing patch identification for stable versions, rather than the whole back-porting scenario. The other two stages, i.e., bug detection & reporting and debugging & bug fixing, are beyond the scope of this thesis.

1.2 Mining Software Repositories

Practitioners like bug triagers, developers, testers often make decisions based on their experience in previous software projects. For instance, bug triagers filter, prioritize, and assign bugs to developers who might be familiar with the concerns related to the bugs. Developers commonly use their experience when adding a new feature or fixing a bug. Testers usually prioritize the testing of features that are known to be error prone based on field and bug reports. Software repositories, such as bug repositories, contain a wealth of valuable information about software. Using the information stored in these repositories, practitioners can depend less on their intuition and experience, and depend more on historical data. In the literature, the line of work on mining software repositories analyzes and cross-links the rich data available in software repositories to help developers make decisions more efficiently or even automatically based on historical data.

In this thesis, we focus on how mining software repositories could help developers automate three tasks in the management of bugs, i.e., bug prioritization, bug assignment, and bug fixing patch identification. Based on the definition of the four-stage bug management process introduced in Section 1.1, the first two tasks happen in the “Bug Triaging” stage while the last task happens in the “Patch Verification & Backporting” stage.

1.3 Outline and Overview

The remainder of this thesis is organized as follows. Chapters 2-5 present our approaches that mine software repositories for automating three tasks, i.e., bug prioritization, bug assignment, and bug fixing patch identification. Chapter 6 provides a literature review of related studies. Chapter 7 summarizes the contributions of this thesis and points to future directions.

1.4 Acknowledgment of Published Work

Most work presented in this thesis have been published in the following international conference proceedings or journals, except for the work presented in Section 5, which will be submitted to a future conference.

- **DRONE: Predicting Priority of Reported Bugs by Multi-Factor Analysis**. This work was published in the proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM 2013). An extended version of this work **Automated Prediction of Bug Report Priority Using Multi-Factor Analysis** was published in the Journal of Empirical Software Engineering (EMSE 2014). The work is presented in Chapter 2.

- **Learning to Rank for Bug Report Assignee Recommendation**. This work was published in the proceedings of the 24th IEEE International Conference on Program Comprehension (ICPC 2016). The work is presented in Chapter 3.

- **Identifying Linux Bug Fixing Patches**. This work was published in the proceedings of the 34th ACM/IEEE International Conference on Software Engineering (ICSE 2012). The work is presented in Chapter 4.

I have published other papers on conference/journal/book/workshop which are not included in this thesis. These papers are either mining software repositories for software maintenance tasks, or mining social media for software engineering. I list them below.

1. **An Exploratory Study of Functionality and Learning Resources of Web APIs on ProgrammableWeb [92]**, 21th International Conference on Evaluation and Assessment in Software Engineering (EASE 2017).

2. **Harnessing Twitter to Support Serendipitous Learning of Developers [80]**, 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2017).

3. **On the Unreliability of Bug Severity Data [91]**, in the Journal of Empirical Software Engineering (EMSE 2016)

4. **What's Hot in Software Engineering Twitter Space? [79]**, 31nd IEEE International Conference on Software Maintenance and Evolution (ICSME 2015 ERA track).

5. **What are the Characteristics of High-Rated Apps? A Case Study on Free Android Applications [98]**, 31nd IEEE International Conference on Software Maintenance and Evolution (ICSME 2015).

6. **A Comparative Study on the Effectiveness of Part-of-Speech Tagging Techniques on Bug Reports [94]**, 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SNEAR 2015 ERA track).

7. **NIRMAL: Automatic Identification of Software Relevant Tweets Leveraging Language Model [78]**, 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2015)

8. **Evaluating Defect Prediction Approaches Using A Massive Set of Metrics: An Empirical Study [110]**, 30th Annual ACM Symposium on Applied Computing (SAC 2015).

9. **Leveraging Web 2.0 for Software Evolution [93]**, book chapter on Evolving Software Systems (ESS 2014).

10. **Potential Biases in Bug Localization: Do They Matter? [37]**, 29th IEEE/ACM International Conference on Automated Software Engineering (ASE 2014).

11. **SEWordSim: software-specific word similarity database [96]**, ACM/IEEE International Conference on Software Engineering (ICSE 2014 Tool track).

12. **BOAT: An Experimental Platform for Researchers to Comparatively and Reproducibly Evaluate Bug Localization Techniques [104]**, ACM/IEEE International Conference on Software Engineering (ICSE 2014 Tool track).

13. **Automated Construction of a Software-Specific Word Similarity Database [95]**, IEEE Software Evolution Week on Software Maintenance Reengineering and Reverse Engineering (CSMR-WCRE 2014).

14. **Predicting Project Outcome Leveraging Socio-Technical Network Patterns [87]**, 17th European Conference on Software Maintenance and Reengineering (CSMR 2013).

15. **Automatic Classification of Software Related Microblogs [71]**, 28th IEEE International Conference on Software Maintenance (ICSM 2012)

16. **Information Retrieval Based Nearest Neighbor Classification for Fine-Grained Bug Severity Prediction [97]**, 19th Working Conference on Reverse Engineering (WCRE 2012).

17. **Observatory of Trends in Software Related Microblogs [3]**, 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012 Tool track).

18. **What Does Software Engineering Community Microblog About? [90]**, 9th Working Conference on Mining Software Repository (MSR 2012).

19. **Improved Duplicate Bug Report Identification [99]**, 15th European Conference on Software Maintenance and Reengineering (CSMR 2012 ERA Track).

Chapter 2

Automated Bug Prioritization via Multi-Factor Analysis

2.1 Introduction

Developers are often overwhelmed with the large number of bug reports. Prioritizing bug reports can help developers manage the bug triaging process better. Developers often leave bug reports unfixed for years due to various factors including time constraints. Thus, it is important for developers to prioritize bug reports well so that important reports are prioritized and fixed first. Bug report prioritization is especially important for large projects that are used by many clients since they typically receive higher numbers of bug reports. Prioritizing bugs is a manual process and is time consuming. Bug triagers need to read the information provided by users in the new bug reports, compare them with existing reports, and decide the appropriate priority levels.

To aid bug triagers in assigning priorities, in this chapter, we propose a new automated approach to recommend priority levels of bug reports. To do so, we leverage information available in the bug reports. Bug reports contain various information including short and long descriptions of the issues users encounter while using the software system, the products that are affected by the bugs, the dates the bugs are reported, the people that report the bugs, the estimated severity of the bugs, and many more. We would like to leverage this information to predict the priority levels of bug reports.

We test our solution on more than a hundred thousand bug reports of Eclipse that span a period of several years. We compare our approach with a baseline solution that adapts an algorithm by Menzies and Marcus [56] for bug priority prediction. Our experiments demonstrate that we can achieve up to 209% improvement in the average F-measure. The contributions of this approach are as follows:

1. We propose a new problem of predicting the priority of a bug given its report. Past studies on bug report analysis have only considered the problem of predicting the severity of bug reports, which is an orthogonal problem.

2. We predict priority by considering the different factors that potentially affect the priority level of a bug report. In particular, we consider the following factors: `temporal`, `textual`, `author`, `related-report`, `severity`, and `product`.

3. We introduce a new machine learning framework, named DRONE, that considers these factors and predicts the priority of a bug given its report. We also propose a new classification engine, named GRAY, which is a component of DRONE, that *enhances* linear regression with *thresholding* to handle imbalanced data.

4. We have tested our solution on more than a hundred thousand bug reports from Eclipse and evaluated its ability to support developers in assigning priority levels to bug reports. The results show that DRONE can outperform a baseline approach, built by adapting a bug report severity prediction algorithm, in terms of average F-measure, with a relative improvement of up to 209%.

Table 2.1: Examples of Bug Reports from Eclipse. Comp.=Component. Sev.=Severity. Prio.=Priority.

	ID	Summary	Product	Comp.	Sev.	Prio.
1	4629	Horizontal scroll bar appears too soon in editor (1GC32LW)	Platform	SWT	normal	P4
	4664	StyledText does not compute correct text width (1GELJXD)	Platform	SWT	normal	P2
2	4576	Thread suspend/resume errors in classes with the “same” name	JDT	Debug	normal	P1
	5083	Breakpoint not hit	JDT	Debug	normal	P1
3	4851	Print ignores print to file option (1GKXC30)	Platform	SWT	normal	P3
	5126	StyledText printing should implement ”print to file”	Platform	SWT	normal	P3

2.2 Background

When a new bug report is submitted into a bug tracking system, a bug triager would first investigate the fields of the bug report and potentially other reports. Based on the investigation, he or she would check the validity of the bug report and may change values of some fields of the bug report. Some bugs are also reported as duplicate bug reports at this point. We show some example bug reports from Eclipse in Table 2.1. Note that bug reports shown in the same box (e.g., 4629 and 4664) are duplicates of one another. Eventually a bug triager would forward the bug to a developer to fix it. The developer then works on the bug and eventually comes up with a resolution. The developer may also change the values of some fields of the bug report when working on it.

2.2.1 Text Pre-processing

In this work, we transform each textual document into a set of features by applying the standard text preprocessing steps, including tokenization, stop-word removal, and stemming [100]. Text preprocessing has two objectives, word normalization and lexicon reduction in a text. We present the detail of each step in the following paragraphs.

Tokenization. A textual document contains many words. Each word is referred to as a token. These words are separated by delimiters which could be spaces, punctuation marks, etc. Tokenization is the process of extracting these tokens from a textual document by splitting the document into tokens at the delimiters.

Stop-Word Removal. Not all words are equally important. There are many words that are frequently used in many documents but carry little meaning or useful information. These words are referred to as stop words. These stop words need to be removed from the set of tokens extracted in the previous steps as they might affect the effectiveness of machine learning or information retrieval solutions due to their skewed distributions. There are many such words, including “am”, “are”, “is”, “I”, “he”, etc. We use a collection of 30 stop words and also standard contractions including, “I’m”, “that’s”, etc.

Stemming. Words can appear in various forms; in English, various grammatical rules dictate whether a root word appear in its singular, plural, present tense, past tense, future tense, or many other forms. Words originating from the same root word but are not identical with one another are semantically related. For example, there is not much difference in meaning between “write” and “writes”. In the text mining and information retrieval community, stemming has been proposed to address this issue. Stemming tries to reduce a word to its *ground* form. For example, “working”, “worked”, and “work” would all be reduced to “work”. Various algorithms have been proposed to perform stemming. In this work, we use the Porter’s stemming algorithm [68] to process the text, as it has commonly been used by many prior studies, e.g., [41, 42, 56, 105].

2.2.2 Measuring the Similarity of Bug Reports

Various techniques have been proposed to measure the similarity of bug reports. A number of techniques model a bug report as a vector of weighted tokens. The similarity of two bug reports can then be evaluated by computing the Cosine similarity of their corresponding two vectors. These include the work by Jalbert and Weimer [29], Runeson et al. [75], Wang et al. [105], etc.

Sun et al. propose an approach called REP, to measure the similarity of bug reports [85]. Their approach extends BM25F [74], which is a state-of-the-art measure for structured document retrieval. In their proposed approach, past bug reports that have been labeled as duplicate are used as training data to measure the similarity of two bug reports. Various fields of bug reports are used for comparison including the textual and non-textual contents of bug reports. We use an adapted version of REP to measure the similarity of bug reports. REP includes the comparison of the priority fields of two bug reports to measure their similarity. In our setting, we would like to predict the values of the priority field. Thus, we remove the priority field from REP's analysis. We call the resulting algorithm REP^{-}. REP^{-} only compares the textual (summary and description), product, and component fields of two bug reports to measure their similarity.

2.3 Problem Definition & Approach

In this section, we first define our problem. Next, we describe our proposed framework. First we present the overall structure of our framework. Next, we zoom into two sub-components of the framework, namely feature extraction and classification modules. In the feature extraction module, we extract various features that capture various factors that potentially affect the priority level of a bug report. In the classification module, we propose a new classification engine leveraging linear regression and thresholding to handle imbalanced data.

2.3.1 Problem Definition

"Given a new bug report and a bug tracking system, predict the priority label of the new report as either P1, P2, P3, P4, or P5."

2.3.2 Approach: Overall Framework

Our framework, named DRONE (PreDicting PRiority via Multi-Faceted FactOr ANalysEs), is illustrated in Figure 2.1. It runs in two phases: training and prediction. There are two main modules: the feature extraction module and the classification module.

In the training phase, our framework takes as input a set of bug reports with known priority labels. The feature extraction module extracts various features that capture temporal, textual, author, related-report, severity, and product factors that potentially affect the priority level of a bug report. These features are then fed to the classification module. The classification module then produces a discriminative model that can classify a bug report with unknown priority level.

In the prediction phase, our framework takes a set of bug reports whose priority levels are to be predicted. Features are first extracted from these bug reports. The model learned in the training phase is then used to predict the priority levels of the bug reports by analyzing these features.

Our framework has two placeholders: the feature extraction and classification modules. Various techniques could be put into these placeholders. We describe our proposed feature extraction and classification modules in the following two subsections.

2.3.3 Feature Extraction Module

The goal of the feature extraction module is to characterize a bug report in several dimensions: temporal, textual, author, related-report, severity,

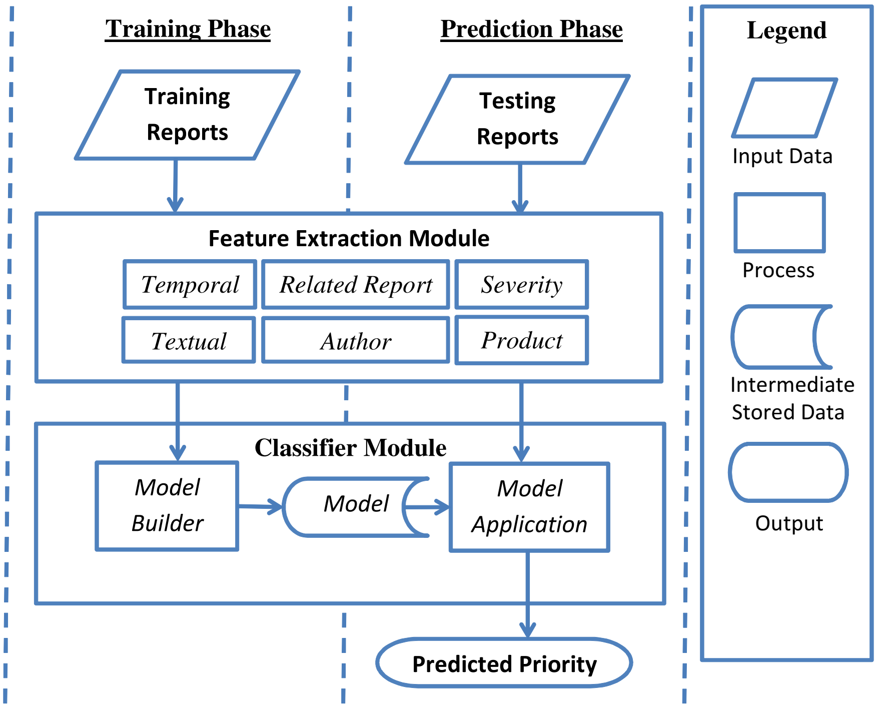


Figure 2.1: DRONE Framework

and `product`. For each dimension, a set of features is considered. For each bug report BR our feature extraction module processes various fields of BR and a bug database of reports created prior to the reporting of BR. It then produces a vector of values for the features listed in Table 2.2 and Table 2.3.

Each dimension/factor is characterized by a set of features. For the `temporal` factor, we propose several features that capture the number of bugs that are reported in the last x days with priority level y. We vary the values of x and y to get a number of features (TMP1-12). Intuitively, if there are many bugs reported in the last x days with a higher severity level than BR, BR is likely not assigned a high priority level since there are many higher severity bug reports in the bug tracking system that need to be resolved too.

For the `textual` factor, we take the description of the input bug report BR and perform the text pre-processing steps listed in Section 2.2. Each of the resulting word tokens corresponds to a feature. For each feature, we take the number of times it occurs in a description as its value. Collectively these features (TXT1-n) describe what the bug is all about and this determines how important it is for a particular bug

Table 2.2: DRONE Features Extracted for a Bug Report BR

	`Temporal` Factor
TMP1	Number of bugs reported within 7 days before the reporting of BR
TMP2	Number of bugs reported with the same severity within 7 days before the reporting of BR
TMP3	Number of bugs reported with the same or higher severity within 7 days before the reporting of BR
TMP4-6	The same as TMP1-3 except the time duration is 30 days
TMP7-9	The same as TMP1-3 except the time duration is 1 day
TMP10-12	The same as TMP1-3 except the time duration is 3 days
	`Textual` Factor
TXT1-n	Stemmed words from the description field of BR excluding stop words (Specifically, n=395,996 in our experiment).
	`Author` Factor
AUT1	Mean priority of all bug reports made by the author of BR prior to the reporting of BR
AUT2	Median priority of all bug reports made by the author of BR prior to the reporting of BR
AUT3	The number of bug reports made by the author of BR prior to the reporting of BR

to get fixed.

For the `author` factor, we capture the mean and median priority, and number of all bug reports that are made by the author of BR prior to the reporting of BR (AUT1-3). We extract `author` factor features based on the hypothesis that if an author always reports high priority bugs, he or she might continue reporting high priority bugs. Also, the more bugs an author reports, it is likely that the more reliable his/her severity estimation of the bug would be.

For the `related-report` factor, we capture the mean and median priority of the top-k reports as measured using REP^-. REP^- is a bug report similarity measure adapted from the studies by Sun et al. [85] – described in Section 2.2. We vary the value k to create a number of features (REP1-10). Considering that similar bug reports might be assigned the same priority, we analyze the top-k most similar reports to a bug report BR to help us decide the priority of BR. For the `severity` factor, we use the severity field of BR as a feature.

Table 2.3: DRONE Features Extracted for a Bug Report BR (Continued)

	`Related-Report` Factor
REP1	Mean priority of the top-20 most similar bug reports to BR as measured using REP^{-} prior to the reporting of BR
REP2	Median priority of the top-20 most similar bug reports to BR as measured using REP^{-} prior to the reporting of BR
REP3-4	The same as REP1-2 except only the top 10 bug reports are considered
REP5-6	The same as REP1-2 except only the top 5 bug reports are considered
REP7-8	The same as REP1-2 except only the top 3 bug reports are considered
REP9-10	The same as REP1-2 except only the top 1 bug report is considered
	`Severity` Factor
SEV	BR's severity field.
	`Product` Factor
PRO1	BR's product field. This categorical feature is translated into multiple binary features.
PRO2	Number of bug reports made for the same product as that of BR prior to the reporting of BR
PRO3	Number of bug reports made for the same product of the same severity as that of BR prior to the reporting of BR
PRO4	Number of bug reports made for the same product of the same or higher severity as those of BR prior to the reporting of BR
PRO5	Proportion of bug reports made for the same product as that of BR prior to the reporting of BR that are assigned priority P1.
PRO6-9	The same as PRO5 except they are for priority P2-P5 respectively.
PRO10	Mean priority of bug reports made for the same product as that of BR prior to the reporting of BR
PRO11	Median priority of bug reports made for the same product as that of BR prior to the reporting of BR
PRO12-22	The same as PRO1-11 except they are for the component field of BR.

For the `product` factor, we capture features related to the product and component fields of BR. The product field specifies a part of the software system that is affected by the issue reported in BR. The component field specifies more specific subparts of the software system that are affected by the issue reported in BR. For each of the product and component fields, we extract 11 features. These product/component features include features that capture the value of the field (PRO1,PRO12), some statistics of bug reports made for that particular product/component prior to the reporting of BR (PRO2-9,PRO13-20), and the mean and median priority levels of bug reports made for that particular product/component prior to the reporting of BR (PRO10-11,PRO21-22). Some products or components might play a more major role in the software systems than other products or components – for these products a triager might assign higher priority levels.

2.3.4 Classification Module

Feature vectors produced by the feature extraction module for the training and testing data are then fed to the classification module. The classification module has two parts corresponding to the training and prediction phases. In the training phase, the goal is to build a discriminative model that can predict the priority of a new bug report with unknown priority. This model is then used in the prediction phase to assign priority levels to bug reports.

In this work, we propose a classification engine named GRAY (ThresholdinG and Linear Regression to ClAssifY Imbalanced Data). We illustrate our classification engine in Figure 2.2. It has two main parts: linear regression and thresholding. Our approach utilizes linear regression to capture the relationship between the features and the priority levels. As our data is imbalanced (i.e., most of the bug reports are assigned priority level P3), we employ a *thresholding* approach to calibrate a set of thresholds to decide the class labels (i.e., priority levels).

We follow a regression approach rather than a standard classification approach

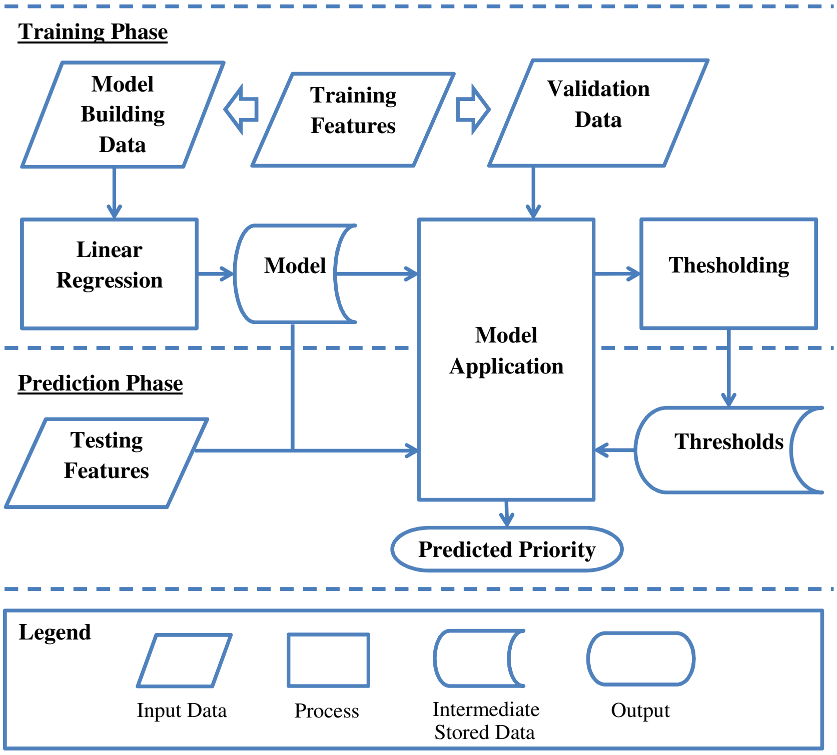


Figure 2.2: GRAY Classification Engine

for the following reason. The bug reports are of 5 priority levels (P1-P5). These priority levels are not categorical values rather they are ordinal values. It means there is an order among these priority levels. For instance, Level P1 is higher than level P2, which is in turn higher than level P3, and so on. Regression makes it possible to capture this ordering among levels. Standard classification approaches, e.g., standard support vector machine, naive bayes, logistic regression, etc., consider the class labels to be categorical. Also, many approaches and standard tools only support two class labels: +ve and -ve.

Given the training data, a linear regression approach builds a model capturing the relationship between a set of explanatory variables with a dependent variable. If the set of explanatory variables has more than one member, it is referred to as multiple regression, which is the case for our approach. In our problem setting, the features form the set of explanatory variables while the priority level is the dependent variable. A bug report in the prediction phase is converted to a vector of feature values, which is then treated as a set of explanatory variables. The model learned during linear regression could then be applied to produce the value for the dependent variable which is a real number.

The next step is to convert the value of the dependent variable to one of the five priority levels. One possibility is to simply truncate the value of the dependent variable to the nearest integer and treat this as the priority level. However, this would not work well for our data as it is imbalanced with most bug reports having priority 3 – thus many of the values of the dependent variable are likely to be close to 3. To address this issue we employ a *thresholding* approach to pick four thresholds to be the boundaries of the five priority levels.

Before performing the thresholding approach, we collect a set of validation data to infer the four thresholds using our thresholding approach. The linear regression model learned from training data is applied on the validation data which generates a priority score for each report. These validation reports with other predicted scores are the input of our thresholding process.

The pseudocode of the thresholding approach which employs greedy hill climbing to tune the thresholds is shown in Algorithm 1. The resulting linear regression model and thresholds are then used to classify bug reports in the testing data whose priority level is to be predicted based on their feature vectors.

We first set the 4 thresholds based on the proportion of bug reports that are assigned as P1, P2, P3, P4, and P5 in the validation data (Line 6). For example, if the proportion of bug reports belonging to P1 in the validation data is only 10%, then we sort the data points in the validation data based on their linear regression scores, and set the first threshold as the regression output of the data point at the 10th percentile. Next, we modify each threshold one by one to achieve a higher F-measure (Lines 8-15). For each threshold level, we try to increase it or decrease it by a small amount, which is 1% of the distance between a threshold level and the previous threshold level (Lines 9, 11-12). At each step, after we change the threshold level, we evaluate whether the resulting threshold levels increase the average F-measure for the validation data points. If it does, we keep the new threshold level, otherwise

Algorithm 1 Tune Thresholds Using Greedy Hill Climbing

```
1: Input:
2: $VData$: Validation Data
3: Output:
4: $T$ : The four thresholds: $T_1$, $T_2$, $T_3$, and $T_4$
5: Method:
6: Initialize $T$ based on the proportion of reports assigned as P1, P2, P3, P4, and P5 in $VData$ (see text).
7: Let $T_0$ = minimum regression score of reports in $VData$.
8: for all $T_i \in \{T_1, T_2, T_3, T_4\}$ do
9:     Let $D = T_i - T_{i-1}$
10:    repeat
11:        Try to increase $T_i$ by $1\% \times D$, compute new F-measure on $VData$
12:        Try to decrease $T_i$ by $1\% \times D$, compute new F-measure on $VData$
13:        Update $T_i$ if the increase or decrease improves F-measure and $T_0 < T_1 < T_2 < T_3 < T_4$
14:    until $T_i$ is not updated
15: end for
16: return Tuned thresholds $T$
```

we discard it (Line 13). We continue the process until we can no longer improve the average F-measure by moving a threshold level, with a constraint that a threshold cannot be moved beyond the next threshold level or under the previous threshold level, i.e., the second threshold cannot be set higher than the third threshold (Line 14).

2.4 Evaluation

2.4.1 Definition of Scenarios

In this section, we first describe the four scenarios in which we apply and evaluate DRONE. We then describe the datasets that we use to investigate the effectiveness of DRONE. Next, we present our experimental setting and evaluation measures. Finally, we present our research questions followed by our findings.

The values of the various fields in a bug report can be changed while the bug report is processed by triagers and developers. Fields can be changed for various reasons. One reason is that the initial values of the fields are incorrect. Based on this observation, we consider four different scenarios:

Last In this scenario, we predict the *last* value of the priority field given the *last* values of other fields in the bug report. We evaluate the effectiveness of our approach when the values of all other fields have been finalized.

Assigned In this scenario, we predict the value of the priority field, given the values of other fields, at the time a bug report status is changed to “Assigned”. When a bug report is received, its status is typically “Unconfirmed” or “New”. After some checks, if it is valid, following standard procedure, its status is eventually changed to “Assigned” indicating that the bug report has been assigned to a developer and the assigned developer is working on the report. At this point, the values of the bug report fields are likely to be more reliable.

First In this scenario, we predict the *first* value of the priority field given the *first* values of other fields in a bug report. This scenario is meant to evaluate how accurate our approach is considering the noisy values of initial bug report fields (i.e., they might get changed later).

No-P3 This scenario is similar to scenario “Last”. The only difference is that we remove all bug reports whose priority levels are “P3” (i.e., the default priority level). Since P3 is the default value of the priority field, it *might* be the case that for P3 bug reports, developers do not put much thought when setting the priority level. However, most bug reports are assigned P3. Thus, deleting these bug reports would mean omitting the majority of bug reports. Due to the pros and cons of excluding (or including) P3 bug reports, we investigate *both* the “Last” and “No-P3” scenarios.

2.4.2 Dataset Collection

We investigate the bug repository of Eclipse. Eclipse is an integrated development platform to support various aspects of software development. It is a large open source project that is supported and used by many developers around the world. In the following paragraphs, we describe how we collect an Eclipse dataset for each of the four scenarios described above.

Scenario “Last”

We consider the bug reports submitted from October 2001 to December 2007 and download them from Bugzilla.[1] We collect only defect reports and ignore those that correspond to feature requests. Since these bug reports were submitted many years back (6-12 years back), the values of various fields in the reports are unlikely to be changed further. These reports contain the last values of the fields after modifications (if any).

[1] https://bugs.eclipse.org/bugs/

Table 2.4: Eclipse Dataset Details. Train.=Training Reports. Test.=Testing Reports.

Period		REP^{-} Train.		DRONE Train.	Test.
From	**To**	**#Duplicate**	**#All**	**#All**	**#All**
2001-10-10	2007-12-14	200	3,312	87,649	87,648

We sort the bug reports in chronological order. We divide the dataset into three: REP^{-} training data, DRONE training data, and the test data. The REP^{-} training data is the first N reports containing 200 duplicate bug reports (c.f. [85]). This data is used to train the parameters of REP^{-} such that it is better able to distinguish similar bug reports. We split the remaining data into DRONE training and testing data. We use the first half of the bug reports (sorted in chronological order) for training and keep the other half for testing. We separate training data and testing data based on chronological order to simulate the real setting where our approach would be used. This evaluation method is also used in many other research studies that also analyze bug reports [24, 61, 75]. We show the distribution of bug reports used for training and testing in Table 2.4.

Scenario “Assigned” and “First”

The datasets used for scenario “Assigned” and “First” are similar to the dataset used for scenario “Last”. However, rather than using the last values of the various fields in the bug reports, we need to reverse engineer the values of the fields when the bug report status was changed to assigned (for Scenario “Assigned”) and the values of the fields when the bug report was submitted (for Scenario “First”). In order to obtain the values of the priority and other fields of bug reports for scenario “First” and “Assigned”, we investigate the modification histories of bug reports. A modification history of a bug report specifies for each modification: the person who made the modification, the time when the modification was performed, the fields whose values get modified, the values that get deleted, and the values that get added.

An example of a modification history of a bug report is shown in Table 2.5.

Table 2.5: Modification History for Bug Report with Id 5110

Who	When	What	Removed	Added
James_Moody	2001-10-26 11:28:58 EDT	Assignee	Kevin_McGuire	James_Moody
James_Moody	2001-10-26 14:21:46 EDT	CC		Kevin_McGuire
James_Moody	2001-11-01 12:07:54 EST	Status	NEW	ASSIGNED
James_Moody	2002-01-03 16:42:56 EST	Priority	P3	P5
Kevin_McGuire	2002-04-17 17:18:09 EDT	Status	ASSIGNED	RESOLVED
		Resolution	—	FIXED
Kevin_McGuire	2002-05-23 21:20:40 EDT	Target Milestone	—	2.0 M6

The modification history specifies that five modifications have been performed. The first modification was performed by James_Moody at 11.28 am EDT on the 26th of October 2001. James_Moody changed the value of the assignee field to himself. The third modification, on the 1st of November 2001, changed the status from new to assigned indicating that he starts working on the bug report. Around two months later, on the 3rd of January 2002, the priority is changed from P3 to P5. This illustrates a bug report where the priority level considered by Scenario “Last” (i.e., P5) differs from the priority level considered by Scenario “Assigned” (i.e., P3).

Another example of a modification history of a bug report is shown in Table 2.6. The modification history specifies that three modifications have been performed. The first modification was performed by paules at 10.35 pm EDT on the 2nd of May

Table 2.6: Modification History for Bug Report with Id 185222

Who	When	What	Removed	Added
paules	2007-05-02 22:35:49 EDT	Status	NEW	ASSIGNED
		Priority	P3	P1
		Target Milestone	—	4.4i3
paules	2007-05-03 08:08:19 EDT	Status	ASSIGNED	RESOLVED
		Resolution	—	FIXED
jptoomey	2007-07-11 12:37:50 EDT	Status	RESOLVED	CLOSED

2007. The developer paules changed two fields: status, summary and target milestone. The status was changed from new to assigned indicating that paules started working on the problem. At the same time, paules changed the value of the priority field from P3 to P1. Also, paules added a target milestone which is 4.4i3. This illustrates a bug report where the priority level considered by Scenario "Last" and "Assigned" (i.e., P1) differs from the priority level considered by Scenario "First" (i.e., P3).

Scenario "No-P3"

The dataset used for scenario "No-P3" is similar to the dataset used for scenario "Last". However, we remove bug reports whose final priority levels are P3 from original DRONE training and testing bug reports. The resulting dataset contains 23,830 bug reports, where 13,529 bug reports are used as training reports and 10,301 bug reports are used as testing reports.

2.4.3 Baseline Approaches

We compare our approach with an adapted version of Severis which was proposed by Menzies and Marcus [56]. Severis predicts the severity of bug reports. In the adapted Severis, we directly use it to predict the priority of bug reports. We use the same feature sets and the same classification algorithm described in the Menzies and Marcus's paper. Following the experimental setting described in their paper, we use the top 100 word token features (in terms of their information gain) as it has been shown to perform best among the other options presented in their paper. We refer to the updated Severis as Severis^{Prio}. We also add the severity label as an additional feature to Severis^{Prio} and refer to the resulting solution as Severis^{Prio+}. We compare Severis^{Prio} and Severis^{Prio+} to our proposed framework DRONE. All experiments are run on an Intel Xeon X5675 3.07GHz server, having 128.0GB RAM, and running the Windows Server 2008 operating system.

2.4.4 Evaluation Measures

We use precision, recall, and F-measure, which are commonly used to measure the accuracy of classification algorithms, to evaluate the effectiveness of DRONE and our baseline approaches: Severis^{Prio} and Severis^{Prio+}. We evaluate the precision, recall, and F-measure for each of the priority levels. This follows the experimental setting of Menzies and Marcus to evaluate Severis [56]. The definitions of precision, recall, and F-measure for a priority level P are given below:

$$prec(P) = \frac{Number\ of\ priority\ P\ reports\ correctly\ labeled}{Number\ of\ reports\ labeled\ as\ of\ priority\ level\ P}$$

$$recall(P) = \frac{Number\ of\ priority\ P\ reports\ correctly\ labeled}{Number\ of\ priority\ P\ reports}$$

$$F\text{-}measure(P) = 2 \times \frac{precision \times recall}{precision + recall}$$

2.4.5 Research Questions

For the first scenario (Scenario "Last"), we consider four research questions:

RQ1 How accurate is our proposed approach as compared with the baseline approaches $Severis^{Prio}$ and $Severis^{Prio+}$?

RQ2 How efficient is our proposed approach as compared with the baseline approaches $Severis^{Prio}$ and $Severis^{Prio+}$?

RQ3 Which of the features are the most effective in discriminating the priority levels?

RQ4 What are the effectivenesses of the various classification algorithms in comparison with GRAY in predicting the priority levels of bug reports?

For the other scenarios, since the answers to RQ2 to RQ4 are likely to be similar to the answers for the first scenario, we only focus on answering RQ1.

2.5 Evaluation Results & Discussion

2.5.1 Results for Scenario "Last"

Here, we present the answers to the four research questions for scenario "Last". The first two compare DRONE with Severis^{Prio} and $Severis^{Prio+}$ on two dimensions: accuracy and efficiency. The best approach must be accurate and yet be able to complete training and prediction fast. Next, we zoom in to the various factors that influence the effectiveness of DRONE. In particular, we inspect the features that are most discriminative. We also replace the classification module of DRONE with several other classifiers and investigate their effects on the accuracy of the resulting approach.

RQ1: Accuracy of DRONE vs. Accuracy of Baselines

The results for DRONE are shown in Table 2.7. We note that we can predict the P1, P2, P3, P4, and P5 priority levels with F-measures of 41.76%, 11.64%, 86.85%, 0.43%, and 8.01% respectively. The F-measures are better for the P1, P2, and P3 priority levels but are worse for the P4, and P5 priority levels.

Table 2.7: Precision, Recall, and F-Measure for DRONE (Scenario "Last")

Priority	**Precision**	**Recall**	**F-Measure**
P1	41.15%	42.39%	41.76%
P2	10.92%	12.46%	11.64%
P3	91.36%	82.77%	86.85%
P4	0.24%	1.77%	0.43%
P5	4.97%	20.72%	8.01%
Average	29.73%	32.02%	29.74%

Table 2.8: Precision, Recall, and F-Measure for Severis^{Prio} (Scenario "Last")

Priority	**Precision**	**Recall**	**F-Measure**
P1	0.00%	0.00%	0.00%
P2	0.00%	0.00%	0.00%
P3	88.25%	100.00%	93.76%
P4	0.00%	0.00%	0.00%
P5	0.00%	0.00%	0.00%
Average	17.65%	20.00%	18.75%

Table 2.9: Precision, Recall, and F-Measure for Severis^{Prio+} (Scenario "Last")

Priority	**Precision**	**Recall**	**F-Measure**
P1	0.00%	0.00%	0.00%
P2	0.00%	0.00%	0.00%
P3	88.25%	100.00%	93.76%
P4	0.00%	0.00%	0.00%
P5	0.00%	0.00%	0.00%
Average	17.65%	20.00%	18.75%

The result for Severis^{Prio} is shown in Table 2.8. We note that Severis^{Prio} can predict the P1, P2, P3, P4, and P5 priority levels by F-measures of 0.00%, 0.00%, 93.76%, 0.00%, and 0.00% respectively. The F-measures of Severis^{Prio} are zeros for P1, P2, P4, and P5 as it does not assign any bug report correctly in any of these priority levels. Comparing these with the result of DRONE (in Table 2.7), we note that we can improve the average of the F-measures by a relative improvement of 58.61% (i.e., $(29.74 - 18.75)/18.75 \times 100\%$). Thus, clearly DRONE performs better than Severis^{Prio}. We believe that in report prioritization higher accuracy for high priority bugs (i.e., P1 and P2) is much more important than higher accuracy for low priority bugs (i.e., P3, P4, and P5) because identifying them can help developers fix the most important bug reports first. We also believe that higher accuracy for bug reports with priority P4 and P5 is more important than higher accuracy for bug reports with priority P3. This is the case since developers expected that they can safely set bug reports with priority P4 and P5 aside (they are unimportant) and fix them when time permits, which can improve the overall efficiency. On the other hand, since the majority of bug reports are P3 reports, developers can neither prioritize them or safely set them aside.

The results for Severis^{Prio+} are shown in Table 2.9. We note that the results of Severis^{Prio+} are the same as the results of Severis^{Prio}. Thus, our proposed approach DRONE also outperforms Severis^{Prio+}.

RQ2: Efficiency of DRONE vs. Efficiency of Baselines

We compare the runtime of DRONE with those of Severis^{Prio} and Severis^{Prio+}. The results are shown in Table 2.10. The four columns refer to the average feature extraction time (for training data), the model building time, the average feature extraction time (for testing data), and the average model application time. We note that the time for feature extraction is slower for DRONE than for the two variants of Severis. Instead, DRONE utilizes more features than the two variants of Severis. Severis^{Prio} only utilizes the textual features of bug reports. Severis^{Prio+} only utilizes the textual and severity features of bug reports. The time for model building, however, is faster for DRONE than for the two variants of Severis. We compare the efficiency of the approaches since the required running time determines the usability of the system for triagers.

RQ3: Most Discriminative Features

Next, we would like to find the most discriminative features among the 20,000+ features that we have (including the word tokens). Information gain [54] and Fisher score [19] are often used as discriminativeness measures. Since many of the features are non-binary features, we use Fisher score as it captures the differences in the

Table 2.10: Efficiency of Severis^{Prio}, Severis^{Prio+}, and DRONE (Scenario "Last"). FE = Average Feature Extraction Time. MB = Model Building Time. MA = Average Model Application Time.

Approach	**Time (in seconds)**			
	FE (Train)	MB	FE (Test)	MA
Severis^{Prio}	<0.01	812.18	<0.01	<0.01
Severis^{Prio+}	<0.01	773.62	<0.01	<0.01
DRONE	0.01	69.25	0.02	<0.01

Table 2.11: Top-10 Features in Terms of Fisher Score (Scenario "Last")

Rank	**Feature Name**	**Fisher Score**
1	PRO5	0.142
2	PRO16	0.132
3	REP1	0.109
4	REP3	0.101
5	PRO18	0.092
6	PRO10	0.091
7	PRO21	0.088
8	PRO7	0.088
9	REP5	0.087
10	"1663"	0.079

distribution of the feature values across the classes (i.e., the priority levels).

At times, features that are only exhibited in a few data instances receive a high Fisher score. This is true for the word tokens. However, these are not good features as they appear too sparsely in the data. Thus we focus on features that appear in at least 0.5% of the data. For these features, Table 2.11 shows the top-10 features sorted according to their Fisher score (the higher the better). We notice that six of them are features related to the `product` factor and three of them are features related to the `related-report` factor. These observations suggest that the product a bug report is about and existing related reports influence the priority label assigned to the report.

We notice that the 10^{th} most discriminative feature is a word token "1663". This token comes from a line in various stack traces included in many bug reports which is:

org.eclipse.ui.internal.Workbench.run(Workbench.java:1663)

It is discriminative as it appears in 15% of the bug reports assigned priority level

P5, while it only appears in 0.77%, 1.29%, 0.99%, and 0.00% of the bug reports assigned priority level P1, P2, P3, and P4 respectively. It seems the inclusion of stack traces that include the above line enables developers to identify P5 bugs better.

RQ4: Effectiveness of Various Classification Algorithms

The classification engine of our DRONE framework could be replaced with other classification algorithms than GRAY. We experiment with several classification algorithms (SVM-MultiClass [15], RIPPER [11], and Naive Bayes Multinomial [54]) and compare their F-measures across the five priority levels of those with GRAY. We use the implementation of SVM-MultiClass available from [88]. We use the implementations of RIPPER and Naive Bayes Multinomial in WEKA [106]. We show the result in Table 2.12. We notice that in terms of average F-Measures GRAY outperforms SVM-MultiClass by a relative improvement of 58.61%. Naive Bayes Multinomial is unable to complete due to an out-of-memory exception although we have allocated more than 9GB of RAM to the JVM in our server. RIPPER could not complete after running for more than 8 hours.

Table 2.12: Comparisons of Average F-Measures of GRAY versus Other Classifiers (Scenario “Last”). Class. = Classifiers. SM = SVM-MultiClass. NBM = Naive Bayes Multinomial. OOM = Out-Of-Memory (more than 9GB). CC = Cannot Complete In Time (more than 8 hours).

Class.	**F-Measures**					
	P1	P2	P3	P4	P5	Ave.
GRAY	41.76%	11.64%	86.85%	0.43%	8.01%	29.74%
SM	0%	0%	93.76%	0%	0%	18.75%
RIPPER	CC	CC	CC	CC	CC	CC
NBM	OOM	OOM	OOM	OOM	OOM	OOM

2.5.2 Results for Scenario “Assigned”

Here, we present the answer to the first research question for the scenario “Assigned”. The result of DRONE is shown in Table 2.13. We note that we can predict the P1, P2, P3, P4, and P5 priority levels with F-measures of 34.34%, 17.13%, 86.20%, 6.19%, and 4.61% respectively. The F-measures are better for the P1, P2,

Table 2.13: Precision, Recall, and F-Measure for DRONE (Scenario “Assigned”)

Priority	Precision	Recall	F-Measure
P1	36.24%	32.64%	34.34%
P2	13.58%	23.19%	17.13%
P3	90.55%	82.25%	86.20%
P4	3.72%	18.52%	6.19%
P5	3.21%	8.18%	4.61%
Average	29.46%	32.96%	29.69%

Table 2.14: Precision, Recall, and F-Measure for Severis^{Prio} (Scenario “Assigned”)

Priority	Precision	Recall	F-Measure
P1	0%	0%	0%
P2	0%	0%	0%
P3	86.27%	99.86%	92.57%
P4	0%	0%	0%
P5	0%	0%	0%
Average	17.25%	19.97%	18.51%

Table 2.15: Precision, Recall, and F-Measure for Severis^{Prio+} (Scenario “Assigned”)

Priority	Precision	Recall	F-Measure
P1	17.07%	0.63%	1.22%
P2	23.33	0.56%	1.09%
P3	86.31%	99.68%	92.51%
P4	0%	0%	0%
P5	0%	0%	0%
Average	25.34%	20.17%	18.96%

and P3 priority levels but are worse for the P4 and P5 priority levels.

The results for Severis^{Prio} are shown in Table 2.14. We note that Severis^{Prio} can predict the P1, P2, P3, P4, and P5 priority levels by F-measures of 0%, 0%, 92.57%, 0%, and 0% respectively. Note that the F-measures of Severis^{Prio} are zeros for P1, P2, P4, and P5 as Severis^{Prio} predicts most of these bug reports as P3 priority level. Comparing these with the result of DRONE (in Table 2.13), we note that we can improve the average of the F-measures by a relative improvement of 60.4%. Thus, DRONE performs better than Severis^{Prio}.

The result for Severis^{Prio+} is shown in Table 2.15. We note that the result of Severis^{Prio+} is a little better than Severis^{Prio}. But the performance of Severis^{Prio+} is still worse than out proposed approach DRONE. DRONE can improve the average

F-measure by a relative improvement of 56.59%.

2.5.3 Results for Scenario "First"

Here, we present the answer to the first research question for the scenario "First". The results of DRONE are shown in Table 2.16. We note that we can predict the P1, P2, P3, P4, and P5 priority levels by F-measures of 0%, 0%, 99.92%, 0%, and 0% respectively. The F-measures of DRONE for P1, P2, P4 and P5 are all zeros because it predicts almost every bug report as being at the P3 priority level.

Table 2.16: Precision, Recall, and F-Measure for DRONE (Scenario "First")

Priority	**Precision**	**Recall**	**F-Measure**
P1	0%	0%	0%
P2	0%	0%	0%
P3	99.99%	99.85%	99.92%
P4	0%	0%	0%
P5	0%	0%	0%
Average	20.00%	19.97%	19.98%

Table 2.17: Precision, Recall, and F-Measure for Severis^{Prio} (Scenario "First")

Priority	**Precision**	**Recall**	**F-Measure**
P1	0%	0%	0%
P2	0%	0%	0%
P3	100%	100%	100%
P4	0%	0%	0%
P5	0%	0%	0%
Average	20%	20%	20%

Table 2.18: Precision, Recall, and F-Measure for Severis^{Prio+} (Scenario "First")

Priority	**Precision**	**Recall**	**F-Measure**
P1	0%	0%	0%
P2	0%	0%	0%
P3	100%	100%	100%
P4	0%	0%	0%
P5	0%	0%	0%
Average	20%	20%	20%

The results for Severis^{Prio} and Severis^{Prio+} are shown in Table 2.17 and Table 2.18. We note that these two approaches have similar results as DRONE. They can predict the P1, P2, P3, P4, and P5 priority levels by F-measures of 0%, 0%, 100%, 0%, and 0% respectively. One reason why the performance of all approaches are worse for scenario “First” is that *almost all* of the bug reports are initialized with priority P3, which is the default value.

2.5.4 Results for Scenario “No-P3”

Here, we present the answer to the first research questions for scenario “No-P3”. The results of DRONE are shown in Table 2.19. We note that we can predict the P1, P2, P4, and P5 priority levels with F-measures of 67.03%, 62.27%, 8.92%, and 54.96% respectively. The F-measures are better for P1, P2 than for P4 and P5.

Table 2.19: Precision, Recall, and F-Measure for DRONE (Scenario “No-P3”)

Priority	**Precision**	**Recall**	**F-Measure**
P1	69.78%	64.49%	67.03%
P2	61.04%	63.56%	62.27%
P4	10.88%	7.56%	8.92%
P5	46.98%	66.21%	54.96%
Average	47.17%	50.46%	48.30%

Table 2.20: Precision, Recall, and F-Measure for Severis^{Prio} (Scenario “No-P3”)

Priority	**Precision**	**Recall**	**F-Measure**
P1	54.17%	0.60%	1.18%
P2	43.93%	99.42%	60.94%
P4	12.50%	0.16%	0.32%
P5	0%	0%	0%
Average	27.65%	25.04%	15.61%

Table 2.21: Precision, Recall, and F-Measure for Severis^{Prio+} (Scenario “No-P3”)

Priority	**Precision**	**Recall**	**F-Measure**
P1	74.33%	26.20%	38.75%
P2	48.76%	90.39%	63.35%
P4	51.87%	31.19%	38.96%
P5	77.78%	0.86%	1.7%
Average	63.18%	37.16%	35.69%

The results for Severis^{Prio} are shown in Table 2.20. We note that Severis^{Prio} can predict the P1, P2, P4, and P5 priority levels with F-measures of 1.18%, 60.94%, 0.32%, and 0% respectively. Comparing these with the results of DRONE (in Table 2.19), we note that DRONE improves the average F-measure by a relative improvement of 209%. Thus, clearly DRONE performs better than Severis^{Prio}.

The result for Severis^{Prio+} is shown in Table 2.21. We note that the result of Severis^{Prio+} is much better than that of Severis^{Prio}. Severis^{Prio+} can predict the P1, P2, P4 and P5 priority levels by F-measures of 38.75%, 63.35%, 38.96%, and 1.7% respectively. We note that our approach DRONE still performs better than Severis^{Prio+}, with a relative improvement of 35%.

2.5.5 Threats to Validity

Threats to construct validity relate to the suitability of our evaluation measures. We use precision, recall, and F-measure, which are standard metrics used for evaluating classification algorithms. Also, these measures are used by Menzies and Marcus to evaluate Severis [56].

Threats to internal validity relate to experimental errors. We have checked our implementation and results. Still, there could be some errors that we did not notice.

Threats to external validity refer to the generalizability of our findings. We consider the repository of Eclipse, which contains more than a hundred thousand bugs that are reported in a period of more than 6 years. Still, we have only analyzed bug reports from one software system. We have excluded some other Bugzilla datasets from two other software systems that we have, as most of the reports there do not contain information in the priority field. In the future, we plan to extend our study by considering more programs and bug reports. In addition, the learned model may not be able to prioritize bug reports from other projects, i.e., the proposed approach does not deal with the cold-start situation where there is little training data for a new project. We will consider using a transfer learning technique in the future work for cross project bug prioritization.

2.6 Chapter Conclusion

In this chapter, we have proposed a framework named DRONE to predict the priority levels of bug reports in Bugzilla. We consider multiple factors including: temporal, textual, author, related-report, severity and product. These features are then fed to a classification engine named GRAY built by combining linear regression with a thresholding approach to address the issue with imbalanced data and to assign priority labels to bug reports. We have compared our approach with several baselines based on the state-of-the art study on bug severity prediction by Menzies and Marcus (2008). The result on a dataset consisting of more than 100,000 bug reports from Eclipse shows that our approach outperforms the baselines in terms of average F-measure by a relative improvement of up to 209% (Scenario “No-P3”).

Chapter 3

Learning-to-Rank for Automatic Bug Assignment

3.1 Introduction

To improve bug triage efficiency and effectiveness, researchers have proposed numerous approaches to automatically assign bug reports to developers, through understanding and extracting useful information from historical bug reports and source code [4, 26, 50, 60, 82, 89]. Shokripour et al. categorized this prior work into two groups, based on their underlying mechanism: activity-based approaches, and location-based approaches [82]. *Activity-based* approaches [4, 60, 89] identify potentially appropriate developers based on their activities (e.g., previously fixed bugs) within the project, across various project artifacts. By contrast, *location-based* approaches [26, 50, 82] recommend a bug report assignment by localizing the bug to a set of potential source code locations and then identifying which developers touched the implicated code. Each approach has its pros and cons. For instance, location-based approaches are highly reliant on the performance of bug localization, which might not be high (c.f. [44, 58, 102, 107]); activity-based approaches might be inappropriately biased by the previous activities of a given developer. We discuss these limitations in more detail in Section 3.2. Importantly, none of the previous work has combined the two types of information, which motivates our study.

In this chapter, we propose a unified model based on the learning to rank machine learning technique that *combines* information from both developers' previous activities and suspicious program locations associated with a bug report in the form of *similarity features*. Learning to rank is a machine learning technique widely applied in applications like document retrieval. We choose learning to rank because we can map the assignee recommendation task to a document retrieval task by treating the bug report as the query, and developers' profiles (previously fixed bugs and committed source code) as documents to be returned. To incorporate location information, the query can be enriched with the potential locations where the bug may reside. This reduces the task to ranking documents (developers) based on the similarity between a query (bug report) and each document (developer profile).

To capture the similarity between a bug report and developer profile, we propose 16 features, considering both the potential location of the bug (location-based features) and previously fixed bugs by each developer (activity-based features). To evaluate our approach, we collect more than 11,000 bug reports together with committed source code from three open source projects: Eclipse JDT, Eclipse SWT and ArgoUML. Our experiments show that combining these two types of features improves the performance of learning to rank model as compared to one that uses only one type of feature. The experiments also show that our unified model achieves better results as compared to a location-based baseline by Anvik et al. [4] and an activity-based baseline by Shokripour et al. [82]. Our key contributions are thus:

1. A novel unified model based on learning to rank machine learning algorithm. This unified model can leverage information from both developers' activities and the result of bug report localization task. This integrates activity-based and location-based bug assignee recommendation approaches.

2. 16 features to capture the degree to which a developer matches a bug report.

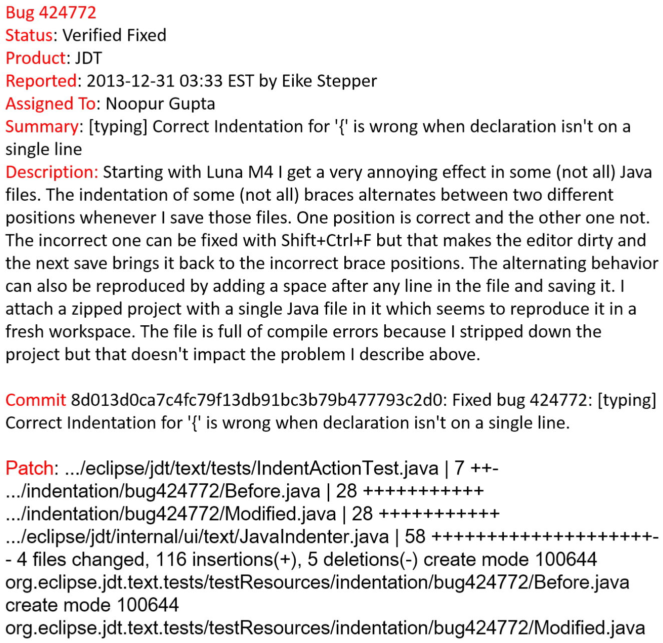
3. Experimental results on more than 11,000 bugs from three open source

projects. The results show that combining location-based features and activity-based features through the learning to rank technique can improve the performance as compared to using only one type of feature.

3.2 Background

In this section, we first introduce basic background on the bug assignee recommendation task. Next, we summarize the two types of automatic bug assignee recommendation approaches that have been considered in prior work. To illustrate this discussion, consider the sample bug report taken from the Bugzilla report database for Eclipse (ID 424772), shown in Figure 3.1. Bugzilla provides several fields to help describe and manage a bug. In Figure 3.1, we list six fields of particular interest to the bug assignee recommendation task: bug **Status**, the **Product** in which the bug appears, the time at which the bug was **Reported**, the developer to whom the bug was assigned (**Assigned To**), and both a short **Summary** and a long **Description** to provide details and describe steps for reproduction. This report in particular describes a typing related bug under text component of JDT. We also know that this bug was fixed by a developer named Noopur Gupta through a commit, ID of which starts with “8d013d”.

When a bug report like this arrives in the system, it usually does not have an assigned developer (like Noopur Gupta). It is the task of the triager or project maintainers to analyze such a report, establish its validity and uniqueness, and then identify the appropriate person or team to address it. This process is manual and time-consuming, given the hundreds of reports a large project receives daily. Our goal in this work is to automate the process of identifying the appropriate developer to whom such a new, valid report should be assigned. The approaches in previous work extract information from such a report and associated project activity recorded in a bug tracking or source control system to construct predictive models. They can be categorized according to the type of information they use: activity-based approaches



Bug 424772
Status: Verified Fixed
Product: JDT
Reported: 2013-12-31 03:33 EST by Eike Stepper
Assigned To: Noopur Gupta
Summary: [typing] Correct Indentation for '{' is wrong when declaration isn't on a single line
Description: Starting with Luna M4 I get a very annoying effect in some (not all) Java files. The indentation of some (not all) braces alternates between two different positions whenever I save those files. One position is correct and the other one not. The incorrect one can be fixed with Shift+Ctrl+F but that makes the editor dirty and the next save brings it back to the incorrect brace positions. The alternating behavior can also be reproduced by adding a space after any line in the file and saving it. I attach a zipped project with a single Java file in it which seems to reproduce it in a fresh workspace. The file is full of compile errors because I stripped down the project but that doesn't impact the problem I describe above.

Commit 8d013d0ca7c4fc79f13db91bc3b79b477793c2d0: Fixed bug 424772: [typing] Correct Indentation for '{' is wrong when declaration isn't on a single line.

Patch: .../eclipse/jdt/text/tests/IndentActionTest.java | 7 ++-
.../indentation/bug424772/Before.java | 28 +++++++++++
.../indentation/bug424772/Modified.java | 28 +++++++++++
.../eclipse/jdt/internal/ui/text/JavaIndenter.java | 58 ++++++++++++++++++++-
- 4 files changed, 116 insertions(+), 5 deletions(-) create mode 100644
org.eclipse.jdt.text.tests/testResources/indentation/bug424772/Before.java
create mode 100644
org.eclipse.jdt.text.tests/testResources/indentation/bug424772/Modified.java

Figure 3.1: Bug report #424772 from Eclipse JDT.

(see Section 3.2.1) rely on developer activities across various artifacts, linked primarily to textual features in the bug report, while location-based approaches (see Section 3.2.2) use the bug report to locate potentially defective source code files, to identify the developers strongly associated with that code (e.g., developers who created the file, developers who modified the file, developers who modified similar files, etc.).

3.2.1 Activity-based Bug Assignee Recommendation

Activity-based approaches recommend a developer for a particular bug report based on how well the developer's expertise is predicted to match with the given bug report. Developer expertise is inferred from developer activities during previous bug triage processes [4, 89] and then linked to the words that appear in a new bug report. Consider the bug report shown in Figure 3.1. The text in the summary and description fields indicates that the problem lies in the JDT Text component, using words like "Indentation", "typing", "braces", "position". Searching the bug database for the JDT project for the keyword "Indentation" reveals several previously fixed bugs related to this concept, and find that many of them were fixed by/assigned to Noopur, the developer that addressed this new bug. For example, Noopur was also assigned to Bug #404821,[1] which reported that the "Code Indentation" feature of the JDT did not work. This illustrates that a report's description and summary can provide useful textual information to suggest developers with expertise in a given problem or concept, based on previously fixed bugs.

Previous researchers leveraged this insight in several ways. For instance, Anvik et al. [4] treat the developer as a class label, and the bug assignee recommendation task as a multi-class classification problem. They extract features from a set of bug reports, i.e., words that appear in the description and summary field of bug report, and represent each bug report as a feature vector. The value of a feature is the number of times a particular word appears in the report. These feature vectors, together with a set of known assignees drawn from previously-addressed bugs, are used as input to learn a predictive model using a classification algorithm (e.g., Support Vector Machine, Naive Bayes Classifiers, Decision Tree). When a new bug report arrives, a similar feature extraction process is applied, and the trained predictive model can be applied on the new feature vector to predict who should fix it.

Although activity-based bug assignee recommendation approaches have been shown to achieve acceptable results, they ignore a valuable source of information, namely the link between bug reports and source code files. This information is leveraged by location based bug assignee recommendation.

3.2.2 Location-based Bug Assignee Recommendation

The underlying idea behind location-based bug assignee recommendation approach is to identify potential developer expertise on the bug report through the source code itself [26, 50, 82]. The basic assumption is that developers who have recently fixed a bug in a given source code file are more likely to have the required expertise to fix a new bug in the same location than other developers. Under this assumption, a

[1] https://bugs.eclipse.org/bugs/show_bug.cgi?id=404821

developer, even one who has been less active in previous bug fixing activities, has substantial expertise in recently-touched or modified code in the repository.

Generally, these approaches consistent of two phases: (1) bug report localization, followed by (2) bug assignee recommendation. For each of the phases, researchers have proposed various approaches. Hossen et al. apply an information retrieval technique, i.e, latent semantic indexing (LSI) [17] to compute the similarity between a given source file and a bug report [26]. They consider words appearing in identifiers and comments extracted from a source code file as an input document and words appearing in the summary and description field of a bug report as a query. Different from Hossen et al., Shokripour et al. compute a relevance score between a bug report and a source code file by summing the weights of each noun that is common between the bug report and file [82]. The weight of a noun is determined based on the number of times the noun appears in a bug report, a commit message, a source code comment and an identifier. For instance, to fix the sample bug shown in Figure 3.1, Noopur committed several source files including one named "/eclipse/jdt/internal/ui/text/JavaIndenter.java". When Shokripour et al. compute the similarity between the sample bug and file "JavaIndenter.java", the word "indentation" has a weight of 3, because it appeared in three information sources, the commit message, the identifiers in the file, and the bug report.

Although location-based approaches consider similarities between bug reports and source code files, which activity based approaches ignore, they have drawbacks:

- **High dependence on an underlying bug localization technique.** Finding the relevant source code files given a bug report is the initial and one of the most important steps for location-based bug assignee recommendation approaches. Therefore, the performance of the bug localization approach used highly impact the performance of a location-based bug assignee recommendation approach. However, bug localization based on a human written report is a hard problem in and of itself, with common accuracy around 30% for predicting the most suspicious source file, e.g., [111].

- **Ignore rich information contained in historical bug reports.** Many location based approaches do not consider the textual information inside previously fixed bug report, which often contain useful information to determine the expertise of a developer.

To summarize, both activity-based and location-based bug assignee recommendation approaches have advantages and disadvantages. In this work, we combine the two to build a unified bug assignee recommendation model that improves on the performance of the previous approaches.

3.3 Approach

In this section, we detail our proposed bug assignee recommendation approach. We first introduce the overall framework of our approach (Section 3.3.1). We then introduce the features that we use to capture the degree of match between a developer and a bug report, which include those derived from activity information (Section 3.3.3) and location-based information (Section 3.3.4). We also describe the process of extracting these features.

3.3.1 Overall Framework

We apply learning to rank to train a ranking model that uses both activity information and location information as features to identify appropriate developers to address a particular bug report. Learning to rank is a popular machine learning technique for training a model to solve a ranking problem. It has been widely used in various applications, such as document retrieval [51, 72]. Document retrieval is a task that takes as input a query, and retrieves and ranks documents based on their degrees of match with the query. This problem is similar to our assignee recommendation problem, where a new bug report is the query, and the profiles built from developers' activity information form the documents. To incorporate location information, the query would be enriched with the potential locations where the bug may reside. In this way, we can naturally apply learning to rank to build ranking models for the bug assignee recommendation problem.

Figure 3.2 shows the general process of our approach. The recommendation system maintains profiles for all available developers, which we refer to as $D_1, \ldots, D_N$. The main task of this recommendation system is to train a ranking model $f(Br_i, D_j)$ that accurately captures the degree to which a given bug report Br_i matches a given developer j's profile (D_j). To train $f(Br_i, D_j)$, this system requires a set of previously fixed bug reports for which we know the developers to whom they were ultimately assigned. Thus, for a set of M training bug reports $Br_1 \ldots Br_M$ and associated developers $D_1, \ldots, D_N$ the system collects a set of features to represent the degree of match (or similarity) between each bug report and developer. For instance, $d_{1,1}$ represents the similarity between Br_1 and D_1. This information is then used to train the ranking model $f(Br_i, D_j)$ using an off-the-shelf implementation of a learning to rank algorithm. Then, when a new bug report arrives, the trained model calculates the similarity between it and all the potential developers, producing $d_{M+1,1}, d_{M+1,2}, \ldots, d_{M+1,N}$. The output of the whole system for this bug report is a ranked list of developers, where developers at the top of the list have higher similarity scores with the given bug report and are thus more likely to be good choices for addressing the defect.

The ranking model $f(Br_M, D_N)$ is represented as a weighted sum of k features, where each feature $\phi_i(Br_M, D_N)$ captures an element of the similarity between the bug report M and developer N:

$$f(Br_M, D_N) = \sum_{i=1}^{k} w_i * \phi_i(Br_M, D_N)$$

The model parameters w_i are learned from the training set by the learning-to-rank algorithm. The learning-to-rank algorithm employs an optimization procedure

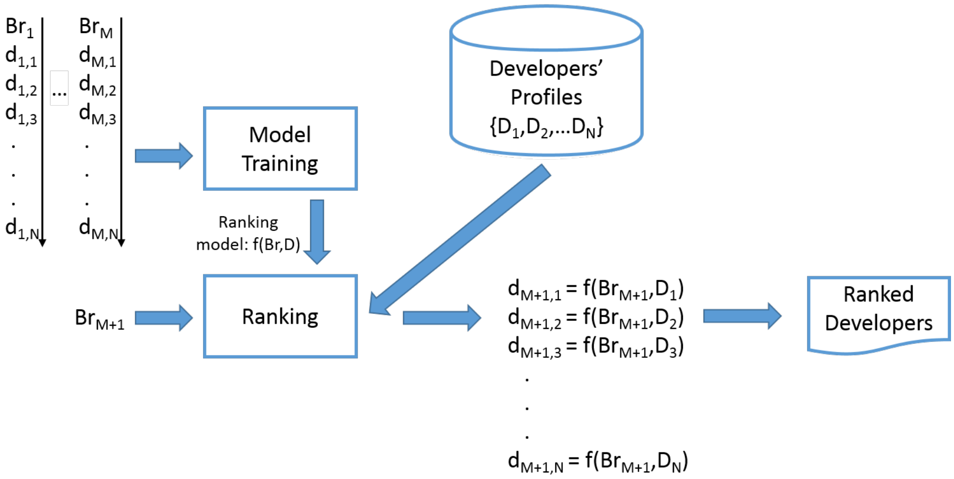


Figure 3.2: Overall Ranking Process

that seeks a set of parameters that results in a function that correctly ranks the developer profiles that are known to be assigned to the bug reports in the training set, at the top of the lists for those bug reports.

In the following sub-sections, we introduce the 16 features, i.e., $\phi_1(Br_M, D_N) \ldots \phi_{16}(Br_M, D_N)$ that we use to measure the degree of match (or similarity) between a bug report and a developers' profile. These features are derived from developers' bug-fixing activities (see Section 3.3.3) and estimated bug locations in the source code (see Section 3.3.4). Table 3.1 summarizes these 16 features.

3.3.2 Dataset Collection and Text Pre-processing

In this work, we consider two kinds of resources to build a developer profile that captures expertise: (1) bug reports that have been fixed by the developer, and (2) the corresponding committed source code files. Here, *corresponding files* refer to the files containing code that has been added, modified, or deleted over the course of fixing corresponding bugs. To complete this task, we first collect a set of fixed bug reports and their links to source files committed to a source control system. In this work, we consider three datasets from the projects Eclipse JDT, Eclipse SWT, and ArgoUML. For Eclipse JDT and SWT, we use the same benchmark dataset provided

Table 3.1: Sixteen Activity-Based and Location-Based Features Characterizing a Bug Report-Developer Pair.

Category	ID	Dimension	Description
Activity-Based	ϕ_{1-5}	Bug Report-Code Similarity	Similarity between source files related to the developer and bug report.
	ϕ_{6-10}	Bug Report-Bug Report Similarity	Similarity between previous bug reports related to the developer and bug report.
	ϕ_{11}	Developer Bug Fixing Frequency	How frequently the developer fixes bugs.
	ϕ_{12}	Developer Bug Fixing Recency	How recently the developer fixes bugs.
Location-Based	ϕ_{13-14}	Potential Buggy Code-Related Code Similarity	Similarity between potential buggy files corresponding to the bug report and source files related to the developer.
	ϕ_{15}	Touched Potential Buggy Files	Whether the potential buggy files have been touched by the developer.
	ϕ_{16}	Touched Mentioned Files	Whether classes mentioned explicitly in bug reports have been touched by the developer.

by Ye et al. [111], where bug reports are already linked with their corresponding bug fixing commits. For ArgoUML, we apply the heuristic approach proposed by Bachmann and Bernstein [5] to link bug reports with commits. Following this approach, we first scan commit logs to find patterns, such as “issue 180”, that could identify bug fixing commits. We then check if the bug reports corresponding to the identifiers exist in the bug tracking system with their status marked as fixed. We also check whether the time the source code files were committed is later than the time the bug report was reported.

After collecting the bug reports and source code files, we extract words appearing in the comments and identifiers of each source code file, and words appearing in the summary and description fields of each bug report. Next, we process the extracted textual information following general textual pre-precessing steps, i.e., tokenization, stop-word removal, and stemming. A token is a string of characters, and includes no delimiters such as spaces, punctuation marks, and so forth. Tokenization is the process of parsing a character stream into a sequence of tokens by splitting the stream at delimiters. Stop words are non-descriptive words carrying little useful information for retrieval tasks. These include verbs such as “is”, “am” and “are”, pronouns such as “I”, “he” and “it”, etc. Our stop word list contains 30 stop words, and also common contractions such as “I’m”, “that’s”, “we’ll”, etc. Stemming is a technique to normalize words to their ground forms. For example, a stemmer can reduce both “working” and “worked” to “work”. We use the Porter stemming algorithm [31] to perform this step.

3.3.3 Extraction of Activity-Based Features

This subsection describes the features mined from developers’ bug fixing activities.

Bug Report-Code Similarity This dimension (ϕ_{1-5}) captures the *textual* similarity between a bug report Br and previous files containing source code committed by a developer D to fix prior bugs. We combine the summary and description fields in a bug report into one document per report. We consider a source file that a developer has *touched* (i.e., added, deleted, or modified) as a document. We also create a merged document that contains all files a developer has touched. We consider two metrics to measure document similarity: cosine and BM25 similarity scores.

To compute the cosine similarity (i.e., $\phi_1(Br, D)$) between a bug report Br and a developer D, we first define a function $Cosine(r, s)$ that calculates the cosine similarity between two documents (in our case, a document could be a bug report, a code file or a merged code file) r and s. Function $Cosine$ first transforms pre-processed words in document r and s into two vectors of weights. Each word is mapped to an element of the vector. The weight of a word $term$ in a vector corresponding to a document doc is computed as:

$$w_{term,doc} = TF_{term,doc} \times IDF_{term}$$

In the above equation, $TF_{term,doc}$ is the number of times word $term$ appears in doc. *IDF*$_{term}$ $= log\frac{N}{DF_{term}}$, where DF_{term} is the number of documents that contain word $term$, given a document corpus. TF-IDF (term frequency - inverse document frequency) is a popular way to assign weights in information retrieval [54]. The vector representations of two documents are then compared by computing their cosine similarity as follows:

$$\textit{Cosine}(r, s) = \frac{\overrightarrow{r} \cdot \overrightarrow{s}}{\|\overrightarrow{r}\| \|\overrightarrow{s}\|} \tag{3.1}$$

In the above equation, $\overrightarrow{r}$ and $\overrightarrow{s}$ are the vector representations of the bug report and the set of patches, $\cdot$ is the dot production of the two vectors, and $\|\overrightarrow{v}\|$ is the size of vector $\overrightarrow{v}$.

BM25 is another way to compute similarity between documents [74]. Given a query q (e.g., a bug report) and a document s (e.g., a document that contains all source code files touched by developer D to fix prior bugs), *BM25*(q, s) computes a similarity score as follows:

$$\textit{BM25}(q, s) = \sum_{i=1}^{n} \textit{Idf}_{q_i} \cdot \frac{f(q_i, s) \cdot (k_1 + 1)}{f(q_i, s) + k_1 \cdot ((1 - b + b \cdot \frac{|s|}{avgdl}))} \tag{3.2}$$

In the above equation, q_i is the ith word in the query q, $f(q_i, s)$ is number of times q_i appears in document s, $|s|$ is the length of the document (i.e., number of words in the document), and $avgdl$ is the average document length in the text collection from which documents are drawn (i.e., average number of words in the documents containing touched source code files of different developers). k_1 and b are free parameters. In our experiment, we set k_1 and b as 1.2 and 0.75, as suggested by Manning et al. [54].

Based on these two types of similarity metrics, we define the following five features:

$$\phi_1(Br, D) = max(\textbf{\textit{Cosine}}(Br, m)|m \in D_{CodeCorpus})$$

$$\phi_2(Br, D) = avg(\textbf{\textit{Cosine}}(Br, m)|m \in D_{CodeCorpus})$$

$$\phi_3(Br, D) = sum(\textbf{\textit{Cosine}}(Br, m)|m \in D_{CodeCorpus})$$

$$\phi_4(Br, D) = \textbf{\textit{Cosine}}(Br, D_{MergedCode})$$

$$\phi_5(Br, D) = \textbf{\textit{BM25}}(Br, D_{MergedCode})$$

In the above equations, Br is the target bug report. $D_{CodeCorpus}$ is the set of source files touched by developer D to fix previous bugs. For ϕ_{1-3}, we consider each source file (m), as a document, and compute their similarity with the bug report. We then use the maximum value, mean, and sum of these similarity scores as the values of the features. For ϕ_{4-5}, we merge all source files to create a larger document $D_{MergedCode}$ for developer D, and compute its similarity with bug report Br using cosine similarity and BM25.

Bug Report-Bug Report Similarity This dimension (ϕ_{6-10}) captures the textual similarity between a bug report Br and all previous bug reports fixed by a developer D. The underlying idea is that words appearing in the bug reports that have been fixed by a developer might capture the expertise of this developer along different aspects. Similar to ϕ_{1-5}, we consider five kinds of similarity metrics in this dimension, given a bug report Br and a developer D:

$$\phi_6(Br, D) = max(\textbf{\textit{Cosine}}(Br, m)|m \in D_{BugCorpus})$$

$$\phi_7(Br, D) = avg(\textbf{\textit{Cosine}}(Br, m)|m \in D_{BugCorpus})$$

$$\phi_8(Br, D) = sum(\textbf{\textit{Cosine}}(Br, m)|m \in D_{BugCorpus})$$

$$\phi_9(Br, D) = \textbf{\textit{Cosine}}(Br, D_{MergedBugs})$$

$$\phi_{10}(Br, D) = \textbf{\textit{BM25}}(Br, D_{MergedBugs})$$

In the above equations, function $Cosine$ and $BM25$ are the same as defined in Equations 3.1 and 3.2 respectively. For the feature ϕ_{6-8}, $D_{BugCorpus}$ is the set of bug reports to which developer D was assigned before Br was reported. For the latter two features, we merge all documents in $D_{BugCorpus}$ as one document, i.e., $D_{MergedBugs}$, and then compute the similarities between two documents.

Developer Bug Fixing Frequency A developer who has fixed a lot of bugs for a project generally has more expertise on the project compared with other developers. Based on this assumption, we consider the number of bugs that have been fixed by a developer over a period of time (one year, in our experiments) as one of the activity-based features. It is defined as:

$$\phi_{11}(Br, D) = |br_{Oneyear}(Br, D)|$$

In the equation above, $br_{Oneyear}(Br, D)$ is the set of bugs that developer D has fixed within one year prior to the reporting of Br.

Developer Bug Fixing Recency Similar to the intuition captured by ϕ_{11}, we speculate that a developer who has recently fixed bugs might more likely to fix a new bug than another developer who has not fixed any bugs in a long time. Let $br(Br, D)$ be the set of bug reports for which developer D has fixed before bug report Br was reported. Let $last(Br, D)$ be the most recently fixed bug in $br(Br, D)$. Also, for any bug report Br, let $Br.date$ denote the date when the bug report was created. We then define the bug-fixing recency feature ϕ_{12} to be the inverse of the distance (in months) between Br and $last(Br, D)$:

$$\phi_{12}(Br, D) = (\textit{diff}^{MTH}(Br.date, last(Br, D).date) + 1)^{-1}$$

In the equation above, *diff*$^{MTH}(Br.date, last(Br, D).date)$ denotes the difference between the dates Br and $last(Br, D)$ were rounded to the nearest number of months.

3.3.4 Extraction of Location-Based Features

Potential Buggy Code-Related Code Similarity To compute these features, we perform two steps: (1) given a bug report, generate a list of source code files that are most likely to be relevant to the bug report using the bug report localization technique proposed by Ye et al. [111], (2) generate features ϕ_{13-15}, to capture the degree of relevance between a developer and a bug report by analyzing the potential location of the bug. We consider the approach proposed by Ye et al. because it is reported to be the state-of-the-art bug report localization technique so far.

ϕ_{13} and ϕ_{14} correspond to the cosine and BM25 similarity scores between the top-k most likely source code files to contain the bug and a document containing all source code files that are touched by a developer to fix prior bugs. They are mathematically defined below:

$$\phi_{13}(Br, D) = MAX_{C_i \in TopK}(\textit{Cosine}(C_i, D_{MergedCode}))$$

$$\phi_{14}(Br, D) = AVG_{C_i \in TopK}(\textit{BM25}(C_i, D_{MergedCode}))$$

In the equation above, $TopK$ refers to a list of top-k files that are most likely to contained the bug described in Br as outputted by Ye et al.'s technique [111]. In the experiment, we set K to 10. $D_{MergedCode}$ is a document that contains all code files touched by D to fix prior bugs.

Touched Potential Buggy Files ϕ_{15} measures whether the developer has touched a file that is potentially buggy when fixing prior bugs. We identify a list of top-K potentially buggy files in a similar way as when we compute ϕ_{13} and ϕ_{14}. In the experiment, we set K to 10 by default.

$$\phi_{15}(Br, D) = \begin{cases} 1, & \text{if developer D has touched } C_i \in \text{TopK} \\ 0, & \text{otherwise} \end{cases} \tag{3.3}$$

Touched Mentioned Files For some of the reported bugs, developers mention the names of some classes (i.e., source code files) in the description of a bug report [37]. These files are likely to be the buggy ones. Thus, we define another feature as follows:

$$\phi_{16}(Br, D) = |Br.files \bigcap D_{CodeCorpus}|$$

In the equation above, $Br.files$ corresponds to the set of source code files whose names appear in Br and $D_{CodeCorpus}$ corresponds to a set of source code files that are touched by D to fix prior bugs.

3.4 Evaluation

In this section, we first present the research questions that we consider in this Chapter. Next, we describe the datasets that we use in this study, followed by our experimental settings. We then present the measures used to evaluate the approaches, followed by our results. Finally, we also mention some threats to validity.

3.4.1 Research Questions

Our core hypothesis is that activity-based and location-based information provide complementary information that can be used to more accurately assign bug reports to developers in a software project. We therefore investigate the following three research questions:

> RQ1: Does a bug assignee prediction model that combines activity-based features and location-based features achieve better performance than a model that uses only one type of feature?

RQ2: Does our unified approach outperform existing activity-based or location-based approaches?

RQ3: Which features are the most important to the accuracy of our model?

For RQ1, we compare three results: the results of our unified model using only activity-based features (i.e., ϕ_{1-12}), using only location-based features (i.e., ϕ_{13-16}), and using all features. We use the learning to rank tool named rankSVM [2] provided by Lee and Kuo to train our unified model.

For RQ2, we consider two baselines:

1. **Activity-baseline:** We use the activity-based approach proposed by Anvik et al. that takes words from the summary and description of bug reports as features and applies the Support Vector Machine (SVM) classifier [4]. Note that this method only returns one label (that is, a single developer) for each bug report.

2. **Location-baseline:** We use the location-based approach proposed by Shokripour et al. [82]. This baseline first uses the weighted sum of common words appearing in bug report and source code file to locate potential files that are related to a bug report. It then recommends a ranked list of assignees based on their recent activity on the potential buggy files.

For the activity-baseline, we use the SVM package in Weka [106] to train SVM classifiers from training data and test it on testing data. For the location-baseline, we write Java code to implement their approach.

For RQ3, we estimate the importance of each features (i.e., ϕ_{1-16}) by considering its corresponding weight w_i (defined in Section 3.3.1), averaged over all training folds.

[2] https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/#large_scale_ranksvm

3.4.2 Dataset

We use bug reports from several open source projects: Eclipse JDT,[3] Eclipse SWT,[4] and ArgoUML.[5]. For the first two datasets, we consider the same set of bug reports as Ye et al. in their paper [111]. These bug reports have been linked to commits that fix them. For ArgoUML, we manually download the bug reports and link them to their corresponding bug fixing commits following the heuristics described by Bachmann and Bernstein [5]. Note that we only consider bug reports with status "fixed" for training and testing. Overall, we consider a total of 11,887 bug reports. Table 3.2 describes the details of the three datasets.

Table 3.2: Datasets: Eclipse JDT, Eclipse SWT, ArgoUML

Project	Time Range	# Bug Reports
JDT	2001-10-10 - 2014-01-14	6,274
SWT	2002-02-19 - 2014-01-17	4,151
ArgoUML	2000-02-01 - 2012-12-13	1,462

3.4.3 Experiment Setup and Evaluation Metrics

As described in Section 3.3, our ranking model $f(Br_M, D_N)$ is based on a weighted combination of features that capture domain dependent relationships between a bug report Br_M and a developer D_N. We train the model parameters w_i using the learning-to-rank approach implemented in the rankSVM package [47].

To mitigate the risk of overfitting, we create disjoint training and test data by sorting the bug reports from each benchmark dataset chronologically by reporting timestamp, because temporal order matters in this data. For instance, we need to make sure that features are extracted from source code and bug reports generated before recommending developers. For all projects, the sorted bug reports are then split into 10 equally sized folds $\{fold_1, fold_2, \ldots, fold_{10}\}$, where $fold_1$ contains the

[3] http://www.eclipse.org/jdt

[4] http://www.eclipse.org/swt

[5] http://argouml.tigris.org/

oldest bug reports while *fold*$_{10}$ consists of the most recently reported bugs. The ranking model is trained on *fold*$_k$ $-$ *fold*$_{(k+5)}$ and tested on *fold*$_{(k+6)}$, for all $1 \leq k \leq 5$. In this way, we collect 5 results for each dataset. Since the folds are arranged chronologically, this means that we always train on the previous existing bug reports. For each bug report in a test fold, testing the model means computing the weighted scoring function $f(r, s)$ for each source code file using the learned weights, and ranking all the files in descending order of their scores. We then check if the correct developer (that is, the developer who actually ultimately repaired the bug in question) appears highly ranked in the output list of developers.

Similar to previous work [82, 89], we use Accuracy@K as an evaluation metric. This metric corresponds to the proportion of top-K recommendations that contain the ground truth developer who assigned to the bug report (as recorded in the bug tracking system). We consider $K =$ 1,2,3,4,5, and 10. For instance, if an assignee recommendation system could successfully identify 30 actual assignees for 100 bug reports at top-1 recommendation, then the value of Accuracy@1 would be 0.3.

3.5 Evaluation Results & Discussion

In this section, we present the results of our experiments in form of answers to research questions 1, 2 and 3. We then discuss threats to validity.

3.5.1 Activity-Based Features vs. Location-Based Features vs. All Features.

In the first research question, we evaluate the efficacy of our unified model for the bug report assignee problem and compare it to models built with each of two types of features alone. The results of our three unified models trained with different sets of features are shown in Table 3.3 and 3.4. From the tables, we note that we can achieve an Accuracy@1 of up to 42%, 45%, and 30% on the JDT, SWT, and ArgoUML datasets, respectively. The unified model (with all features) outperforms the other split models in all cases, supporting our claim that there is value in combining activity-based and location-based features in this domain. For Accuracy@1, using all features improves on the results of using activity-based features alone by 12.5%-31.2%, and the results of using location-based features alone by 15.4%-25.0%.

Table 3.3: Results of Our Unified Model Trained with Various Features on Eclipse JDT, Eclipse SWT, and ArgoUML Data

Project	Feature	Acc@1	Acc@2	Acc@3
JDT	Activity Only	32%	58%	72%
	Location Only	35%	57%	69%
	All	42%	65%	79%
SWT	Activity Only	40%	62%	77%
	Location Only	39%	60%	73%
	All	45%	66%	80%
ArgoUML	Activity Only	26%	29%	32%
	Location Only	24%	27%	30%
	All	30%	35%	41%

Table 3.4: Results of Our Unified Model Trained with Various Features on Eclipse JDT, Eclipse SWT, and ArgoUML Data (Continue)

Project	Feature	Acc@4	Acc@5	Acc@10
JDT	Activity Only	85%	90%	96%
	Location Only	78%	83%	89%
	All	89%	93%	97%
SWT	Activity Only	89%	92%	97%
	Location Only	81%	86%	92%
	All	90%	94%	98%
ArgoUML	Activity Only	38%	41%	52%
	Location Only	36%	39%	44%
	All	45%	50%	56%

3.5.2 Our Unified Model Vs Baselines

In the second research question, we compare our unified model to state-of-the-art techniques that use activity- vs. location-based features alone. The results of our model and two selected baselines on the three datasets are shown in Table 3.5 and 3.6. From the tables, we note that in most of the cases, our unified model with all features achieves the best results. The activity baseline consistently performs worst in all cases. The location baseline performs better than our model in two cases, i.e., Accuracy@3 on JDT and SWT dataset, and Accuracy@4 on ArgoUML dataset. For Accuracy@1, our unified model can outperform the activity-based baseline by Anvik and Murphy by 50.0%-100.0%, and the location-based baseline by Shokripour et al. by 11.1%-27.0%.

Table 3.5: Results of Our Approach and Baselines on Eclipse JDT, Eclipse SWT, ArgoUML

Project	Feature	Acc@1	Acc@2	Acc@3
JDT	Activity Baseline	28%		
	Location Baseline	33%	57%	82%
	All	42%	65%	79%
SWT	Activity Baseline	25%		
	Location Baseline	36%	60%	81%
	All	45%	66%	80%
ArgoUML	Activity Baseline	15%		
	Location Baseline	27%	30%	39%
	All	30%	35%	41%

Table 3.6: Results of Our Approach and Baselines on Eclipse JDT, Eclipse SWT, ArgoUML (Continue)

Project	Feature	Acc@4	Acc@5	Acc@10
JDT	Activity Baseline			
	Location Baseline	88%	89%	92%
	All	89%	93%	97%
SWT	Activity Baseline			
	Location Baseline	88%	91%	93%
	All	90%	94%	98%
ArgoUML	Activity Baseline			
	Location Baseline	47%	50%	52%
	All	45%	50%	56%

3.5.3 Importance of Features

In our third research question, we analyze our model to identify which features are most helpful to the assignee recommendation process. For each dataset, we consider the average weight of each feature returned by rankSVM tool when building the prediction model. We select the top-5 features for each dataset; they are shown in Table 3.7.

The top-5 features consist of both location-based features and activity-based features. Feature ϕ_{15} (whether a developer has touched a potential buggy file) consistently ranks the first among all features on the three data sets. On the contrary, ϕ_{16}, which refers to how many times a developer has touched a source file directly mentioned in the bug report, does not appear in any top-5 list. Other features, such as ϕ_3 (i.e., sum of bug report-code cosine similarities) and ϕ_{12} (i.e., developer bug fixing recency) also rank highly when considering the three datasets. Comparing bug report-bug report similarity features (ϕ_{6-10}) and bug report-source code similarity features (ϕ_{1-5}), the latter are slightly more important with slightly more features appearing in the top-5 lists. Overall, we note that the models built for all three datasets include features from both types of data included in the approach.

Table 3.7: Top-5 Most Important Features

	JDT	SWT	ArgoUML
Top-1	ϕ_{15}	ϕ_{15}	ϕ_{15}
Top-2	ϕ_{12}	ϕ_3	ϕ_3
Top-3	ϕ_3	ϕ_{12}	ϕ_7
Top-4	ϕ_7	ϕ_{11}	ϕ_{12}
Top-5	ϕ_{14}	ϕ_7	ϕ_1

3.5.4 Threats to Validity

Threats to construct validity relate to the suitability of our evaluation metrics. Like previous work [82, 60, 4, 89, 50, 26], we consider the developer who has fixed the bug report as the correct assignee for each testing bug report. We consider Accuracy@K as the metric, which is commonly used in previous work in this space [82, 89] and attempts to capture the degree to which our model achieves their stated goals (accurate prediction of which developer should tackle a given report). Threats to internal validity relate to potential errors in our experiments. We have checked our code, but there might still be errors that we did not notice. Threats to external validity refer to the generalizability of our findings. In our experiments, we consider more than 11,000 bug reports from Eclipse JDT, Eclipse SWT, and ArgoUML. Experiments on these datasets show that our unified model performs better when it combines both activity-based and location-based information. It also outperforms two exiting baselines. To further mitigate these threats to external validity, we plan to experiment with more bug reports from more projects in the future.

3.6 Chapter Conclusion

In this chapter, we propose a unified model based on the learning to rank technique to automatically recommend developers to address particular bug reports. The unified model naturally combines location-based information and activity-based information extracted from historical bug reports and source code for more accurate recommendation. We propose 16 similarity features to capture the similarity between a bug report and a developer profile. We evaluate our unified model on a set of more than 11,000 bug reports from several open source projects: Eclipse JDT, Eclipse SWT and ArgoUML. Our experiments show that combining the two types of features (activity-based and location-based) improves the performance of our unified model as compared to when only one type of features is used. The experiments also show that our unified model performs the best when compared to a location-based baseline by Anvik et al. [4] and an activity-based baseline by Shokripour et al. [82]. Among the 16 features we proposed, we find that feature ϕ_{15} (whether a developer has touched a potential buggy file) is the most important feature in our unified model on all of the three data sets. Feature ϕ_3 (i.e., sum of bug report-code cosine similarities) and ϕ_{12} (i.e., developer bug fixing recency) are the second and third most important features.

Chapter 4

Identifying Linux Bug Fixing Patches

4.1 Introduction

For an operating system, reliability and continuous evolution to support new features are two key criteria governing its success. However, achieving one is likely to adversely affect the other, as supporting new features entails adding new code, which can introduce bugs. In the context of Linux development, these issues are resolved by regularly releasing versions that include new features, while periodically fixing some versions for longterm support. The primary development is carried out on the most recent version, and relevant bug fixes are backported to the longterm code.

A critical element of the maintenance of the longterm versions is thus the identification of bug fixing patches. In the Linux development process, contributors submit patches to subsystem maintainers, who approve the submissions and initiate the process of integrating the patch into the coming release. Such a maintainer may also forward the patch to the maintainers of the longterm versions, if the patch satisfies various guidelines, such as fixing a real bug, and making only a small number of changes to the code. This process, however, puts an extra burden on the subsystem maintainers and thus necessary bug fixing patches could be missed. Thus, a technique that could automatically label a commit as a bug fixing patch would be valuable.

In the literature, there are furthermore many studies that require the identification of links between commits and bugs. These include work on empirical study of software changes [57, 83], bug prediction [35, 62], bug localization [16, 52, 55, 73], and many more. All of these studies employ a keyword-based approach to infer commits that correspond to bug fixes, typically relying on the occurrence of keywords such as "bug" or "fix" in the commit log. Some studies also try to link software repositories with a Bugzilla by the detection of a Bugzilla number in the commit log. Unfortunately these approaches are not sufficient for our setting because:

1. Not all bug fixing commit messages include the words "bug" or "fix"; indeed, commit messages are written by the initial contributor of a patch, and there are few guidelines as to their contents.

2. Linux development is mostly oriented around mailing lists, and thus many bugs are found and resolved without passing through Bugzilla.

Some of these limitations have also been observed by Bird et al. [6] who performed an empirical study that show bias could be introduced due to missing linkages between commits and bugs. In view of the above limitations, there is a need for a more refined approach to automatically identify bug fixing patches.

In this chapter, we perform a dedicated study on bug fixing patch identification in the context of the Linux kernel. The results of our study can also potentially benefit studies that require the identification of bug fixes from commits. We propose a combination of text analysis of the commit log and code analysis of the code change to identify bug fixing patches. We use an analysis plus classification framework that consists of: 1) the extraction of basic "facts" from the text and code that are then composed into features; 2) The learning of an appropriate model using machine learning and its application to the detection of bug fixing commits.

A challenge for our approach is to obtain appropriately labeled training data. For positive data, i.e., bug fixing patches, we can use the patches that have been applied to previous Linux longterm versions, as well as patches that have been developed based on the results of bug-finding tools. There is, however, no corresponding set of independently labeled negative data, i.e., non bug fixing patches. To address this problem, we propose a new approach that integrates two learning algorithms via ranking and classification. We have tested our approach on commits from the Linux kernel code repository, and compare our results with those of the keyword-based approach employed in the literature. We can achieve similar precision with improved recall; our approach's precision and recall are 0.601 and 0.875 while those of the key-word based approach are 0.613 and 0.603. Our contributions are as follows:

1. We identify the new problem of finding bug fixing patches to be integrated into a Linux "longterm" release.
2. We propose a new approach to identifying bug fixing patches by leveraging both textual and code features. We also develop a suitable machine learning approach that performs ranking and classification to address the problem of unavailability of a clean negative dataset (i.e., non bug fixing patches).
3. We have evaluated our approach on commits in Linux and show that our approach can improve on the keyword-based approach by up to 45.11% recall while maintaining similar precision.

4.2 Background

Linux is an open-source operating system that is widely used across the computing spectrum, from embedded systems, to desktop machines, to servers. From its first release in 1994 until the release of Linux 2.6.0 in 2003, two versions of the Linux kernel were essentially maintained in parallel: stable versions for users, receiving only bug-fixing patches over a number of years, and development versions, for developers only, receiving both bug fixes and new features. Since the release of Linux 2.6.0, there has been only a single version, which we refer to as the mainline kernel, targeting both users and developers, which includes both bug fixes and new features as they become available. Since 2005, the rate of these releases has been roughly one every three months.

The current frequent release model is an advantage for both Linux developers and Linux users because new features become quickly available and can be tested by the community. Nevertheless, some kinds of users value stability over support for new functionalities. Nontechnical users may prefer to avoid frequent changes in their working environment, while companies may have a substantial investment in software that is tuned for the properties of a specific kernel, and may require the degree of security and reliability that a well-tested kernel provides. Accordingly, Linux distributions often do not include the latest kernel version. For end users, the current stable Debian distribution (squeeze) and the current Ubuntu Long Term Support distribution (lucid) rely on the Linux 2.6.32 kernel, released in December 2009. For industrial users, the same kernel is at the basis of the current versions of Suse Enterprise Linux, Red Hat Enterprise Linux and Oracle Unbreakable Linux.

In recognition of the need for a stable kernel, the Linux development community maintains a "stable" kernel in parallel with the development of the next version, and a number of "longterm" kernels that are maintained over a number of years. For simplicity, in the rest of this Chapter, we refer to both of these as stable kernels. Stable kernels only integrate patches that represent bug fixes or new device identifiers, but no large changes or additions of new functionalities.[1] Such a strategy is possible because each patch is required to perform only one kind of change.[2] Developers and maintainers may identify patches that should be included in the stable kernels by forwarding the patches to a dedicated email address. These patches are

[1] linux-2.6.39/Documentation/stable kernel rules.txt

[2] linux-2.6.39/Documentation/SubmittingPatches.txt

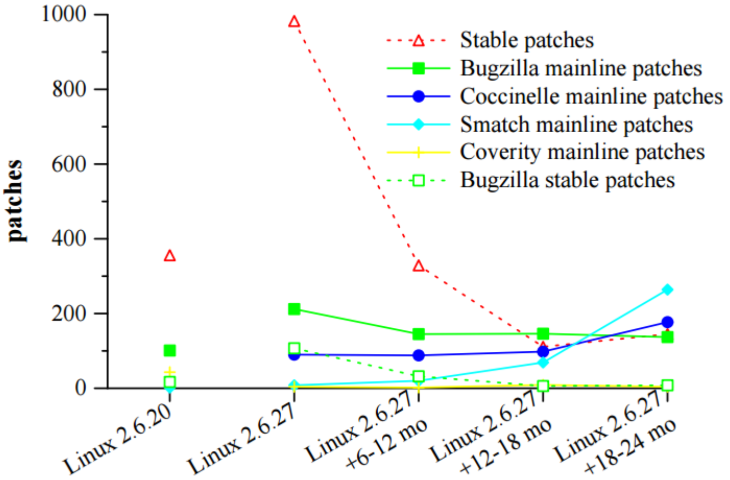


Figure 4.1: Various kinds of patches applied to the stable kernels 2.6.20 and 2.6.27 and to the mainline kernel in the same time period.

then reviewed by the maintainers of the stable kernels before being integrated into the code base.

A comparison, shown in Figure 5.5 of the patches accepted into the mainline kernel with those accepted into the stable kernels Linux 2.6.20, maintained between February 2007 and August 2007, and Linux 2.6.27, maintained between October 2008 and December 2010, shows a gap between the set of bug fixing patches accepted into the mainline as compared to the stable kernels. Specifically, we consider the mainline patches that mention Bugzilla, or that mention a bug finding tool (Coccinelle [63], Smatch [3] or Coverity [4]). We also include the number of patches mentioning Bugzilla that are included in the stable kernel. These amount to at best around half of the Bugzilla patches. Fewer than 5 patches for each of the considered bug finding tools were included in the stable kernel in each of the considered time periods. While it is ultimately the stable kernel maintainers who decide whether it is worth including a bug-fixing patch in a stable kernel, the very low rate of propagation of the considered types of bug-fixing patches from the mainline kernel to the stable kernels suggests that automatic identification of bug fixing patches could be useful.

[3] http://smatch.sourceforge.net/

[4] http://coverity.com/

4.3 Approach

Our approach is composed of the following steps: data acquisition, feature extraction, model learning, and bug-fixing patch identification. These steps are shown in Figure 4.2.

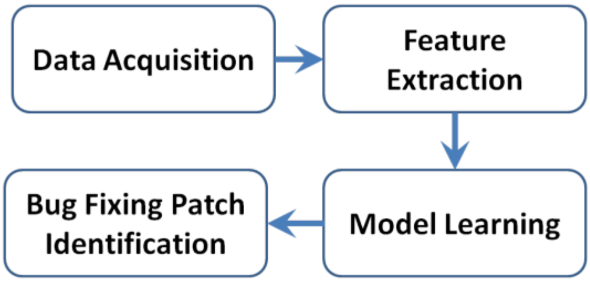


Figure 4.2: Overall Framework

The data acquisition step extracts commits from Linux code repository. Some of these commits represent bug fixing patches while others do not. Not all bug fixing patches are well marked in Linux code. Furthermore, many of these bug fixes are not recorded in Bugzilla. Thus, they are hidden in the mass of many other commits that do not perform bug fixing. There are a variety of non bug fixing commits including those that perform: code cleaning, feature addition, performance enhancement, etc.

The feature extraction component then reduces the dataset into some potentially important facets. Each commit contains a textual description along with code elements that are changed by the commit. The textual description can provide hints whether a particular commit is fixing a bugs or is it only trying to clean up some bad coding style or poor programming practice. Code features also can help a lot. Many bug fixes involve a single location change, boolean operators in if and loop expressions, etc. Many non-bug fixing commits involve substantially many lines of code, etc. To obtain a good collective discriminative features we would need to leverage both text and code based features.

Next, the extracted features are provided to a model learning algorithm that analyzes the features corresponding to bug fixing patches and tries to build a model that discriminates bug fixing patches from other patches. Various algorithms have been proposed to learn a model given a sample of its behavior. We consider some popular classification algorithms (supervised and semi-supervised) and propose a new framework that merges several algorithms together. The final step is the application of our model on the unlabeled data to obtain a set of bug fixing patches.

A challenge in our work is to obtain adequate training data, consisting of known bug fixing patches and known non bug fixing patches. As representatives of bug fixing patches, we may use the patches that have already been applied to Linux stable versions, as well as patches that are known to be bug fixing patches, such as those that are derived from the use of bug finding tools or that refer to Bugzilla. But there is no comparable source of labeled non bug fixing patches. Accordingly, we propose a hybrid machine learning algorithm, that first uses a ranking algorithm to identify a set of patches that appear to be quite distant from the set of bug fixing patches. These patches can then be considered to be a set of known non bug fixing patches. We then use a supervised classification algorithm to infer a model that can discriminate bug fixing from non bug fixing patches in the unlabeled data.

We describe the details of our framework in the following two subsections.

4.3.1 Data Acquisition

Linux development is managed using the version control system git.[5] Git makes available the history of changes that have been made to the managed code in the form of a series of patches. A patch is a description of a complete code change, reflecting the modifications that a developer has made to the source code at the time of a commit. An example is shown in Figure 4.3. A patch consists of two sections: a log message, followed by a description of the code changes. Our data acquisition tool collects information from both of these sections. The collected information is represented using XML, to facilitate subsequent processing.

The log message of a patch, as illustrated by lines 1-16 of Figure 4.3, consists

[5] http://git-scm.com

```
 1  commit 45d787b8a946313b73e8a8fc5d501c9aea3d8847
 2  Author: Johannes Berg <johannes.berg@intel.com>
 3  Date:  Fri Sep 17 00:38:25 2010 +0200
 4
 5      wext: fix potential private ioctl memory content leak
 6
 7      commit df6d02300f7c2fbd0fbe626d819c8e5237d72c62 upstream.
 8
 9      When a driver doesn't fill the entire buffer, old
10      heap contents may remain, . . .
11
12      Reported-by: Jeff Mahoney <jeffm@suse.com>
13      Signed-off-by: Johannes Berg <johannes.berg@intel.com>
14      Signed-off-by: John W. Linville <linville@tuxdriver.com>
15      Signed-off-by: Greg Kroah-Hartman <gregkh@suse.de>
16
17  diff --git a/net/wireless/wext.c b/net/wireless/wext.c
18  index d98ffb7..6890b7e 100644
19  --- a/net/wireless/wext.c
20  +++ b/net/wireless/wext.c
21  @@ -947,7 +947,7 @@ static int ioctl_private_iw_point(. . .
22          } else if (!iwp->pointer)
23                  return -EFAULT;
24
25  -       extra = kmalloc(extra_size, GFP_KERNEL);
26  +       extra = kzalloc(extra_size, GFP_KERNEL);
27          if (!extra)
28                  return -ENOMEM;
29
```

Figure 4.3: A bug fixing patch, applied to stable kernel Linux 2.6.27

of a commit number (SHA-1 code), author and date information, a description of the purpose of the patch, and a list of names and emails of people who have been informed of or have approved of the patch. The data acquisition tool collects all of this information.

The code change of a patch, as illustrated by lines 17-29 of Figure 4.3, appears in the format generated by the command diff, using the "unified context" notation [53]. A change may affect multiple files, and multiple code fragments within each file. For each modified file the diff output first indicates the file name (lines 17-20 of Figure 4.3) and then contains a series of hunks describing the changes (lines 21-29 of Figure 4.3). A hunk begins with an indication of the affected line numbers, in the old and new versions of the file, which is followed by a fragment of code. This code fragment contains context lines, which appear in both the old and new versions, removed lines, which are preceded by a - and appear only in the old version, and added lines, which are preceded by a + and appear only in the new version. A hunk typically begins with three lines of context code, which are followed by a sequence of zero or more removed lines and then the added lines, if any, that replace them. A hunk then ends with three more lines of context code. If changes occur close together, multiple hunks may be combined into a single one. The example in Figure 4.3 contains only a single hunk, with one line of removed code and one line of added code.

Given the different information in a patch, our data acquisition tool records the boundaries between the information for the different files and the different hunks. Within each hunk, it distinguishes between context, removed, and added code. It does not record file names or hunk line numbers.

A commit log message describes the purpose of the change, and thus can potentially provide valuable information as to whether a commit represents a bug fix. To mechanically extract information from the commit logs, we represent each commit log as a bag of words. For each of these bags of words, we perform stop-word removal and stemming [76]. Stop words, such as, is, are, am, would, etc, are used very frequently in almost all documents thus they provide little power in discriminating if a commit is a bug fixing patches or not. Stemming reduces a word to its root; for example, eating, ate, and eaten, are all reduced to the root word eat. Stemming is employed to group together words that have the same meaning but only differ due to some grammatical variations. This process can potentially increase the discriminative power of root words that are good at differentiating bug fixing patches from other commits, as more commits with logs containing the root word and its variants can potentially be identified and associated together after stemming is performed.

At the end of this analysis, we represent each commit as a bag of words, where each word is a root word and not a stop word. We call this information the textual facts that represent the commit.

To better understand the effect of a patch, we have incorporated a parser of patches into our data acquisition tool [64]. Parsing patch code is challenging, because a patch often does not represent a complete, top-level program unit, and indeed portions of the affected statements and expressions may be missing, if they extend beyond the three lines of context information. Thus, the parsing is necessarily approximate. The parser is independent of the line-based - and + annotations, only focusing on the terms that have changed. In the common case of changes in function calls, it furthermore detects arguments that have not changed, counting these separately and ignoring their subterms. For example, in the patch in Figure 4.3, the change is detected to involve a function call, i.e. the call to kmalloc, which is replaced by a call to kzalloc. The initialization of extra is not included in the change, and the arguments to kmalloc and kzalloc are detected to be identical.

Based on the results of the parser, we collect the numbers of various kinds of constructs such as function headers, loops, conditionals, and function calls that include removed or added code. We call these the code facts that represent the commit.

4.3.2 Feature Extraction

Based on the textual and code facts extracted as described above, we pick interesting features that are compositions of several facts (e.g., the difference between the number of lines changed in the minus and plus hunks, etc.).

Table 4.1 presents some features that we form based on the facts. Features F_1 to F_{52} are those extracted from code facts. The other features (i.e., features F_{53} to F_{55} and features W_1 to W_n) are those extracted from textual facts.

For code features, we consider various program units changed during a commit including, files, hunks, loops, ifs, contiguous code segments, lines, boolean operators, etc. For many of these program units, we consider the number of times they are added or removed; and also, the sum and difference of these numbers. Often bug fixing patches, and other commits (e.g., feature additions, performance enhancements, etc) have different value distributions for these code features.

For text features, we consider stemmed non-stop words appearing in the logs as features. For each feature corresponding to a word, we take its frequency or the number of times it appears in a commit log as its corresponding feature value. We also consider two composite families of words each conveying a similar meaning:

Table 4.1: Extracted Features

ID	**Feature**
F_1	Number of files changed in a commit
F_2	Number of hunks in a commit
F_3	#Loops Added
F_4	#Loops Removed
F_5	$\|F_3 - F_4\|$
F_6	$F_3 + F_4$
F_7	$F_{13} > F_{14}$
$F_8 - F_{12}$	Similar to F_3 to F_7 for #Ifs
$F_{13} - F_{17}$	Similar to F_3 to F_7 for #Contiguous code segments
$F_{18}F_{22}$	Similar to F_3 to F_7 for #Lines
$F_{23}F_{27}$	Similar to F_3 to F_7 for #Character literals
$F_{28}F_{32}$	Similar to F_3 to F_7 for #Paranthesized expressions
$F_{33}F_{37}$	Similar to F_3 to F_7 for #Expressions
$F_{38}F_{42}$	Similar to F_3 to F_7 for #Boolean operators
$F_{43}F_{47}$	Similar to F_3 to F_7 for #Assignments
$F_{48}F_{52}$	Similar to F_3 to F_7 for #Function calls
F_{53}	One of these words exists in the commit log robust, unnecessary, improve, future, anticipation, superfluous, remove unused
F_{54}	One of these words exists in the commit log must, needs to, has to, dont, fault, need to, error, have to, remove
F_{55}	The word “warning” exists in the commit log
$W_1 - W_n$	Each feature represents a stemmed non-stop word in the commit log. Each feature has a value corresponding to the number of times the word appears in the commit (i.e., term frequency).

one contains words that are likely to relate to performance improvement, feature addition, and clean up; another contains words that are likely to relate to a necessity to fix an error. We also consider the word “warning” (not stemmed) as a separate textual feature.

4.3.3 Model Learning

We propose a solution that integrates two classification algorithms: Learning from Positive and Unlabeled Examples (LPU) [48][6] and Support Vector Machine (SVM) [13].[7] These learning algorithms take in two datasets: training and testing, where each dataset consists of many data points. The algorithms each learn a model from the training data and apply the model to the test data. We first describe the differences between these two algorithms.

LPU performs semi-supervised classification. Given a positive dataset and an unlabelled dataset, LPU builds a model that can discriminate positive from negative data points. The learned model can then be used to label data with unknown labels. For each data point, the model outputs a score indicating the likelihood that the unlabeled data is positive. We can rank the unlabeled data points based on this score.

SVM on the other hand performs supervised classification. Given a positive dataset and a negative dataset, SVM builds a model that can discriminate between them. While LPU only requires the availability of datasets with positive labels, SVM requires the availability of datasets with both positive and negative labels.

LPU tends to learn a weaker discriminative model than SVM. This is because LPU takes in only positive and unlabeled data, while SVM is able to compare and contrast positive and negative data. To be able to classify well, we propose a combination of LPU and SVM. First, we use LPU to rank how far an unlabeled data point

[6] http:
www.cs.uic.edu/ liub/LPU/LPU-download.html

[7] http://svmlight.joachims.org

is from the positive training data (in descending order). For this, we sort the data points based on their LPU scores, indicating the likelihood of a data point being positive. The bottom k data points, where k is a user-defined parameter, are then taken as a proxy for the negative data. These negative data along with the positive data are then used as the input to SVM. The sequence of steps in our model learning process is shown in Figure 4.4.

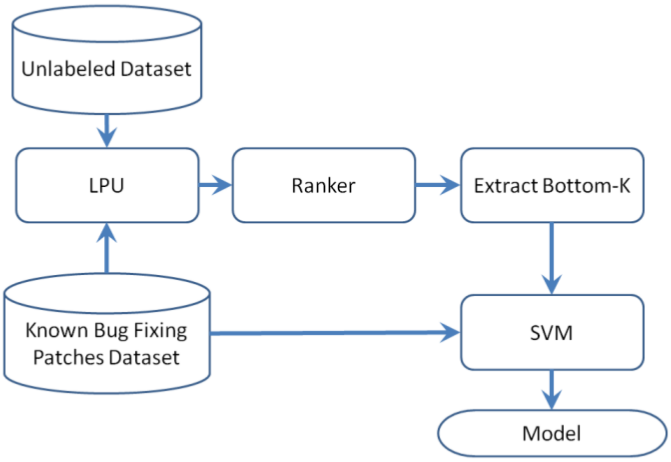


Figure 4.4: Model Learning

In the problem of identifying bug fixing patches, each data point is a commit. We have a set of positive data points, i.e., bug fixing patches, and a set of unlabeled data points, i.e., arbitrary commits. We first apply LPU to sort commits such that bug fixing patches are listed first and other patches, which may correspond to innocuous changes, performance improvements or feature additions, are listed later. According to this ordering, the bottom k commits are likely to be non-bug fixing patches. We then take the bottom k commits to be a proxy of a dataset containing non-bug fixing patches. We use the original bug fixing patch dataset and this data to create a model using SVM.

4.3.4 Bug Fix Identification

For bug fix identification, we apply the same feature extraction process to a test dataset with unknown labels. We then represent this test dataset by a set of feature values. These feature values are then fed to the learned model as described in Section 4.3.3. Based on these features, the model then assigns either one of the following two labels to a particular commit: bug-fixing patch or non bug-fixing patch. [8]

4.4 Evaluation

4.4.1 Dataset

Our algorithm requires as input "black" data that is known to represent bug-fixing patches and "grey" data that may or may not represent bug-fixing patches. The "grey" data may contain both "black" data and "white" data (i.e., non bug-fixing patches).

As there is no a priori definition of what is a bug-fixing patch in Linux, we have created a selection of black data sets from varying sources. One source of black data is the patches that have been applied to existing stable versions. We have considered the patches applied to the stable versions Linux 2.6.20,[9] released in February 2007 and maintained until August 2007, and Linux 2.6.27,[10] released in October 2008 and maintained until December 2010. We have taken only those patches that refer somewhere to C code, and where the code is not in the Documentation section of the kernel source tree. Another source of black data is the patches that have been created based on the use of bug finding tools. We consider uses of the commercial tool Coverity,[11] which was most actively used prior to 2009, and the open source tools Coccinelle [63] and Smatch,[12] which have been most actively used since 2008 and 2009, respectively [65]. The Coverity patches are collected by searching for patches that mention Coverity in the log message. The Coccinelle and Smatch patches are

[8] We use the analogy of black, white and grey in the remaining parts of the paper

[9] http://www.kernel.org/pub/scm/linux/kernel/git/stable/linux-2.6.20.y

[10] http://www.kernel.org/pub/scm/linux/kernel/git/stable/linux-2.6.27.y

[11] http://www.coverity.com

[12] http://smatch.sourceforge.net

Table 4.2: Properties of the considered black datasets. LOC refers to the complete patch size, including both the log and the changed code

Source	**Dates**	**# Patches**	**LOC**
Stable 2.6.20	02.2007 - 08.2007	409	29K
Stable 2.6.27	10.2008 - 12.2010	1534	116K
Coverity	05.2005 - 06.2011	478	22K
Coccinelle	11.2007 - 08.2011	825	54K
Smatch	12.2006 - 08.2011	721	31K
Bugzilla	08.2005 - 08.2011	2568	275K

Table 4.3: Properties of the considered grey dataset, broken down by Linux version. LOC refers to the complete patch size, including both the log and the changed code.

Source	**Dates**	**# Patches**
2.6.20-2.6.21	02.2007-04.2007	3415
2.6.21-2.6.22	04.2007-07.2007	3635
2.6.22-2.6.23	07.2007-10.2007	3338
2.6.23-2.6.24	10.2007-01.2008	4639
2.6.24-2.6.25	01.2008-04.2008	6110
2.6.25-2.6.26	04.2008-07.2008	5069

collected by searching for patches from the principal users of these tools, which are the second author of this paper and Dan Carpenter, respectively. The Coccinelle data is somewhat impure, in that it contains some patches that also represent simple refactorings, as Coccinelle targets such changes as well as bug fixes. The Coverity and Smatch patches should contain only bug fixes. All three data sets are taken from the complete set of patches between April 2005 and August 2011. Our final source of black data is the set of patches that mention Bugzilla. These are also taken from the complete set of patches between April 2005 and August 2011. Table 4.2 summarizes various properties of these data sets.

The grey data is taken as the complete set of patches that have been applied to the Linux kernel between version 2.6.20 and 2.6.26. To reduce the size of the dataset, we take only those patches that can apply without conflicts to the Linux 2.6.20 code base. Table 4.3 summarizes various properties of the data sets.

4.4.2 Research Questions & Evaluation Metrics

In our study, we address the following four research questions (RQ1-RQ4). In RQ1, we investigate the effectiveness of our approach. Factors that influence our effectiveness are investigated in RQ2 and RQ3. Finally, RQ4 investigates the benefit of our hybrid classification model.

RQ1: *Is our approach effective in identifying bug fixing patches as compared to the existing keyword-based method?*

We evaluate the effectiveness of our approach as compared with existing keyword-based method. We consider the following two criteria:

Criteria 1: Precision and Recall on Sampled Data. We randomly sample 500 commits and manually assign labels to them, i.e., each commit is labeled as being either a bug fixing patch or not. We compare human assigned labels with the labels assigned by each bug fix identification approach, and compute the associated precision and recall to evaluate the effectiveness of the approach [76].

Criteria 2: Accuracy on Known Black Data. We take commits that have been identified by Linux developers as bug fixing patches. We split this dataset into ten equal sized groups. We train on 9 groups and use one group to test. We evaluate how many of the bug fixing patches are correctly labeled. The process is iterated 10 times. For each iteration we compute the number of bug fixing patches that are correctly identified (we refer to this as $accuracy^{Black}$) and report the average accuracy.

In the first criteria, our goal is to estimate the accuracy of our approach on some sampled data points. One of the authors is an expert on Linux development and has contributed many patches to Linux code base. This author manually assigned labels to these sampled data points. In the second criteria, we would like to address the experimenter bias existing in the first criteria. Unfortunately, we only have known black data. Thus, we evaluate our approach in terms of its accuracy in labeling black data as such.

RQ2: *What is the effect of varying the parameter k on the results?*

Our algorithm takes in one parameter k, which specifies the number of bottom ranked commits that we take as a proxy of a dataset containing non-bug fixing patches. As a default value in our experiments, we fix this value k to be 0.9 × the number of "black" data that are known bug fixing patches. We would like to vary this number and investigate its impact on the performance.

RQ3: *What are the best features for discriminating if a commit is a bug fixing patches?*

Aside from producing a model that can identify bug fixing patches, we are also interested in finding discriminative features that could help in distinguishing bug fixing patches and other commits. We would like to identify these features out of the many textual and code features that we extract from commits.

We create a clean dataset containing all the known black data, the manually labeled black data, and the manually labeled white data. We then compute the Fisher score [14] of all the features that we have. A variant of Fisher score reported in [10] and implemented in LibSVM[13] is computed. Fisher score and its variants have been frequently used to identify important features [9].

RQ4: *Is our hybrid approach (i.e., ranking + supervised classification using LPU+SVM) more effective than a simple semi-supervised approach (i.e., LPU)?*

Our dataset only contains positively labeled data points (i.e., bug fixing patches). To solve this problem, researchers in the machine learning community have investigated semi-supervised learning solutions. Many of these techniques still required a number of negatively labeled data points. However, LPU [48], which is one of the few semi-supervised classification algorithms with an implementation available online, only requires positively labeled and unlabeled data points.

Our proposed solution includes a ranking and a supervised classification component. The ranking component makes use of LPU. Thus, it is interesting to investigate if the result of using LPU alone is sufficient or whether our hybrid approach could

[13] `http://www.csie.ntu.edu.tw/~cjlin/libsvm/`

Table 4.4: Precision and Recall Comparison

Approach	Precision	Recall
Ours	0.601	0.875
Keyword	0.613	0.603

improve the results of LPU.

4.5 Evaluation Results & Discussion

We present our experimental results as answers to the four research questions: RQ1-RQ4.

4.5.1 Effectiveness of Our Approach

We compare our approach to the keyword-based approach used in the literature [35, 57]. The result of the comparisons using the two criteria are discussed below.

Precision and Recall on Sampled Data. The precision and recall of our approach as compared to those of the keyword-based approach are shown in Table 4.4. We notice that our precision is comparable with that of the keyword-based approach. On the other hand, we increase the recall of the keyword-based approach from 0.603 to 0.875; this is a relative improvement of 45.11%.

To combine precision and recall, we also compute the F-measure [76], which is a harmonic mean of precision and recall. The F-measure is often used as a unified measure to evaluate whether an improvement in recall outweighs the reduction in precision (and vice versa). The F-measure has a parameter β that measures the importance of precision over recall. The formula is:

$$\frac{(\beta^2 + 1) \times precision \times recall}{(\beta^2 \times precision) + recall}$$

In the case that precision and recall are equally important β is set to one. This would compute what is known as F1. If β is set higher than 1 then recall is preferred

Table 4.5: F-Measures Comparison

Approach	**F1**	**F2**	**F3**	**F5**
Ours	0.712	0.802	0.837	0.860
Keyword	0.608	0.605	0.604	0.600
Rel.Improvement	17.11%	32.56%	38.58%	43.33%

Table 4.6: Comparison of $Accuracy^{Black}$ Scores

Approach	**Accuracy**^{Black}
Ours	0.945
Keyword	0.772

over precision; similarly, if β is set lower than 1 then precision is preferred over recall.

In the setting of bug fixing patch identification, recall (i.e., not missing any bug fixing patch) is more important than precision (i.e., not reporting wrong bug fixing patch). This is the case as missing bug fixing patch could potentially cause system errors and even expose security holes. There are also other studies that recommend setting β equal to 2, e.g. [84].

In Table 4.6 we also compute the different F-measures using different values of β. We notice that for all values of β our approach has better results as compared to those of keyword-based approach. The F1, F2, F3, and F5 scores are improved by 17.11%, 31.56%, 38.58%, and 43.33% respectively.

From the 500 randomly sampled commits, we notice that a very small number of commits that are bug fixing patches contains a reference to Bugzilla. Thus, identifying bug fixing patches is not trivial. Also, as shown in Table 4.4, about 40% of bug fixing patches do not contain the keywords considered in previous work [57, 35].

Accuracy on Known Black Data. Table 4.6 shows the $Accuracy^{Black}$ score of our approach as compared to that of keyword-based approach.

From the result, we note that our approach can increase $Accuracy^{Black}$ from 0.772 to 0.945, a 22.4% increase. The above results show that our approach is effective in identifying bug fixing patches as compared to the keyword-based approach used in existing studies.

Table 4.7: Effect of Varying k on Performance. TP = True Positive, FN = False Negative, FP = False Positive, TN = True Negative.

k	**TP**	**FN**	**FP**	**Prec.**	**Recall**	**F2**
0.75	176	8	186	0.486	0.957	0.801
0.80	172	12	166	0.509	0.935	0.801
0.85	168	16	146	0.535	0.913	0.800
0.90	161	23	107	0.601	0.875	0.802
0.95	133	51	68	0.662	0.723	0.710

The known black data is unbiased as we do not label it ourselves. However, we can not compute the number of false positives, as all our test data are black.

The high accuracy of the keyword-based approach is due to the large number of Bugzilla patches in our bug fixing patch dataset. In practice, however, most bug fixing patches are not in Bugzilla. These bug fixing patches are hidden in the mass of other non bug fixing related commits.

4.5.2 Effects of Varying Parameter k

When we vary the parameter k (as a proportion of the number of “black” data), the number of false positives and false negatives changes. The results of our experiments with varying values for k is shown in Table 4.7.

We notice that as we increase the value of k the number of false negatives (FN) increases, while the number of false positives (FP) decreases. As we increase the value of k, the “pseudo-white” data (i.e., the bottom k commits in the sorted list after ranking using LPU) gets “dirtier” as more “black” data are likely to be mixed with “white” data in it. Thus, more and more “black” data are wrongly labeled as “white” (i.e., an increase in false negatives). However, the white data are still closer to the “dirty” “pseudo-white” data than to the black data. Also, more and more borderline “white” data are “closer” to the “dirtier” “pseudo-white” data than before. This would reduce the number of cases where “white” data are labeled “black” (i.e., a reduction in false positives). We illustrate this in Figure 4.5.

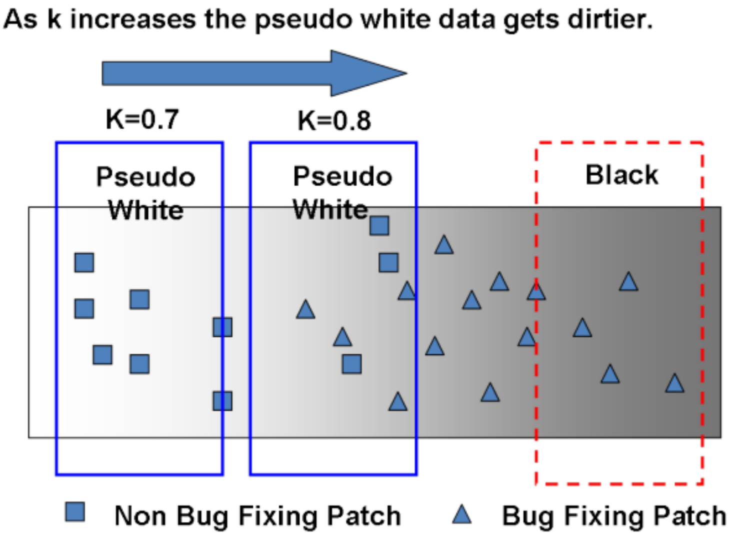


Figure 4.5: Effect of Varying K. The pseudo white data is the bottom k commits that we treat as a proxy to non bug fixing patches. The three boxes corresponding to pseudo white (2 of them) and black represent the aggregate features of the respective pseudo-white and black data in our training set respectively. The squares and triangles represent test data points whose labels (i.e., bug fixing patches or not) are to be predicted.

4.5.3 Best Features

We report the top 20 features sorted based on their Fisher scores in Table 4.8. We note that among the top-20 features there are both textual and code features. This highlight the usefulness of combining both textual features in commit logs and code features in changed code to predict bug fixing patches. We notice however that the Fisher score is low (the highest possible value is 1), which highlights that one feature alone is not sufficient to discriminate positive from negative datasets (i.e., bug fixing patches versus other commits).

Some keywords used in previous approaches [57, 35, 83], e.g., fix, bugzilla, etc., are also included in the top-20 features. Due to tokenization some of these features are split into multiple features, e.g., http, bug.cgi, and bugzilla.kernel.org.

Many other features in the list are code features; these include the number of times different program elements are changed by a commit. The most discriminative code element is the number of lines of code being deleted (ranked 7th). Next include features such as the number of lines added and deleted (ranked 11th), the number of

Table 4.8: Top-20 Most Discriminative Features Based on Fisher Score

Rank	**Feature Desc.**	**Fisher Score**
1	http	0.030
2	blackfin	0.023
3	bug.cgi	0.021
4	show	0.019
5	fix	0.015
6	bugzilla.kernel.org	0.014
7	F_18 (i.e., # lines removed)	0.014
8	commit	0.013
9	upstream	0.012
10	unifi	0.012
11	F_{20} (i.e., # lines added & removed)	0.011
12	id	0.011
13	F_{38} (i.e., # boolean operators removed)	0.011
14	checkpatch	0.011
15	F_{44} (i.e., # assignments removed)	0.010
16	spell	0.010
17	F_{46} (i.e., # assign. removed & added)	0.009
18	F_{37} (i.e., # boolean operators added)	0.009
19	F_6 (i.e., # loops added & removed)	0.009
20	F_{48} (i.e., # function calls added)	0.008

boolean operators added (ranked 13th), the number of assignments removed (ranked 15th), the number of assignments added and removed (ranked 17th), the number of boolean operators added (ranked 18th), the number of loops added and removed (ranked 19th), and the number of function calls made (ranked 20th).

4.5.4 Our Approach versus LPU

We have run LPU on our dataset and found that the results of using LPU alone are not good. The comparison of our results and LPU alone is shown in Table 4.9.

We notice that the precision of LPU is slightly higher than that of our approach; however, the reported recall is much less than ours. Our approach can increase the recall by more than 3 times (i.e., 200% improvement). When we trade off precision and recall using F-measure, we notice that for all β our approach is better than LPU by 78%, 151.4%, 179.0%, and 197.6% for F1, F2, F3, and F5 respectively.

The $accuracy^{Black}$ of our approach and that of LPU alone is comparable. Notice

Table 4.9: Comparisons with LPU

Approach	**Precision**	**Recall**	**F1**
Ours	0.601	0.875	0.712
LPU Only	0.650	0.283	0.400
Rel.Improvement	-7.5%	209.2%	78%

Approach	**F2**	**F3**	**F5**	**Accuracy**^{Black}
Ours	0.802	0.837	0.860	0.944
LPU Only	0.319	0.300	0.289	0.942
Rel.Improvement	151.4%	179.0%	197.6%	0.2%

that the black data in $accuracy^{Black}$ are similar to one another, with many having the terms Bugzilla, http, etc. The black data in the 500 random sample is more challenging and better reflect the black data that are often hidden in the mass of other commits.

The above highlights the benefit of our hybrid approach of combining ranking and supervised classification to address the problem of unavailability of negative data points (i.e., the non bug fixing patches) as compared to a simple application of a standard semi-supervised classification approach. In our approach, LPU is used for ranking to get a pseudo-negative dataset and SVM is used to learn the discriminative model.

4.5.5 Threats to Validity

Threats to internal validity relate to the relationship between the independent and dependent variables in the study. One relevant threat to internal validity in our study is experimenter bias. In the study, we have personally labeled each commits as a bug fixing patch or as a non bug fixing patch. This labeling might introduce some experimenter bias. However, we have tried to ensure that we label the commits correctly, according to our substantial experience with Linux code [43, 63, 64]. Also, we have labeled the commits before seeing the results of our identification approach, to minimize this bias.

Threats to external validity relate to the generalizability of the result. We have manually checked the effectiveness of our approach over 500 commits. Although 500 is not a very large number, we believe it is still a good sample size. We plan to reduce this threat to external validity in the future by investigating an even larger number of manually labeled commits. We have also only investigated patches in Linux. We believe our approach can be easily applied to identify bug fixing patches in other systems. We leave the investigation as to whether our approach remains effective for other systems as future work.

Threats to construct validity relate to the appropriateness of the evaluation criteria. We use the standard measures precision, recall, and F-measure [54] to evaluate the effectiveness of our approach. Thus, we believe there is little threat to construct validity.

4.6 Chapter Conclusion

Linux developers periodically designate a release as being subject to longterm support. During the support period, bug fixes applied to the mainline kernel need to back ported to these longterm releases. This task is not trivial as developers do not necessarily make explicit which commits are bug fixes, and which of them need to be applied to the longterm releases. To address this problem, we propose an automated approach to infer commits that represent bug fixing patches. To do so, we first extract features from the commits that describe those code changes and commit logs that can potentially distinguish bug fixing patches from regular commits. A machine learning approach involving ranking and classification is employed. Experiments on Linux commits show that we can improve on the existing keyword-based approach, obtaining similar precision and improved recall, with a relative improvement of 45.11%.

Chapter 5

Identifying Patches for Linux Stable Versions: Could Convolutional Neural Networks Do Better?

5.1 Introduction

The Linux kernel follows a two-tiered release model in which a *mainline* version, accepting bug fixes and feature enhancements, is paralleled by a series of *stable* versions that accept only bug fixes. The mainline serves the needs of users who want to take advantage of the latest features, while the stable versions serve the needs of users who value stability, or cannot upgrade their kernel due to hardware and software dependencies. To ensure that there is as much review as possible of the bug fixing patches and to ensure the highest quality of the mainline itself, the Linux kernel requires that all patches applied to the stable versions pass through the mainline first. A mainline subsystem maintainer may identify a patch as a bug fixing patch appropriate for stable kernels and add to the commit log a "Cc stable" tag.[1] Stable kernel maintainers then extract such annotated commits from the mainline commit history and apply the resulting patches to the stable versions that are affected by the

[1] The exact tag is `Cc: stable@vger.kernel.org`.

bug.

The quality of the stable kernels critically relies on the effort that the subsystem maintainers put into labeling patches as relevant to the stable kernels, i.e., identifying stable patches. This manual effort represents a potential weak point in the development process – subsystem maintainers could forget to label some relevant patches, and different subsystem maintainers could apply different criteria for selecting them. While the stable maintainers can themselves additionally pick up relevant patches from the mainline commits, there are hundreds of such commits per day, making it likely that many will slip pass. This task can thus benefit from automated assistance.

Previous work presented in Chapter 4 has presented an LPU (Learning from Positive and Unlabeled Examples) and SVM (Support Vector Machine) based approach to automatically identify bug fixing patches (for stable versions). This LPU+SVM based approach relies on 55 features extracted from code changes and thousands of word features extracted from the commit logs. All of the code features are defined manually to characterize how likely it is that a given patch is a bug fixing patch. However, the manual creation of features might overlook good features that could help developers identify bug fixing patches. In addition, relationships between words are ignored, as the LPU+SVM approach considers a bag-of-word representation of text.[2] Thus, a richer feature representation of a patch that can naturally capture its inherent and relevant properties by considering both its commit log and corresponding code changes is needed.

Inspired by recent applications of deep learning techniques in software engineering, we propose a Convolution Neural Network (CNN) based approach to automatically learn features from the commit log and code changes inside a patch for identifying stable (related) patches. To investigate whether the CNN-based approach could perform better than the LPU+SVM based approach, we ask the maintainers of Linux stable versions to help us evaluate the performance of two approaches on

[2] Bag-of-words represents a text as the multi-set of the words that appear in it.

recent Linux patches.

Considering that code is different from natural language content, our CNN-based approach processes code changes separately from the commit log by taking program structure into consideration. The processed code changes and commit log are merged to form a document. Documents generated from a set of training patches are then fed into a Convolution Neural Network based model. The CNN-based model learns network parameters, as well as a classifier, from the training patches. For a new patch, the trained CNN will map the patch into a set of feature-value pairs and predict whether the patch is a bug fixing patch by applying the learned classifier. Our evaluation shows that both the LPU+SVM based and the CNN-based approach have the potential of catching patches that should be considered for stable verions, but have been missed by the maintainers. However, despite the huge benefit of using CNN on some tasks in the Natural Language Processing and Computer Vision domains, our evaluation finds that the CNN-based approach only achieves performance similar to the previous LPU+SVM approach, although it does not require any hand-crafted features.

The main contributions of this work include:

1. We propose a new Convolutional Neural Network (CNN) based approach to identify patches that should be moved to Linux stable versions. Our approach treats the commit log and code changes separately. While we adopt standard natural language processing strategies for representing the commit log, we propose a novel representation of code that incorporates high and low level aspects of the code structure.

2. We take a closer look at the manual process of identifying patches for Linux stable versions. We have summarized the challenges faced by all machine learning approaches in the automation of this manual process.

3. Our experiments are done on a new dataset that contains 48,920 (training and testing) recent Linux patches. We ask the real practitioners, i.e., the maintain-

ers of the Linux stable versions, to evaluate the performance of the new CNN-based approach and the previous LPU+SVM approach. Feedback from the Linux stable kernel maintainers on 199 unique Linux patches shows that the CNN-based approach can achieve a similar performance as the LPU+SVM based approach. Both of these approaches could thus potentially help maintainers find missing patches for stable versions. We also find that these two approaches do show some complementarity, which sheds light on the benefit of combining two approaches.

5.2 Background

In this section, we first present some background information about the maintenance of Linux kernel stable versions, and some challenges that the maintenance of Linux kernel stable versions poses for automation via machine learning. We then present the convolution neural network (CNN) that is considered in our CNN-based approach.

5.2.1 Context

Linux kernel development is carried out according to a hierarchical model, with Linus Torvalds at the root, who has ultimate authority about which patches are accepted into the kernel, and patch authors at the leaves. A patch *author* is anyone who wishes to make a contribution to the kernel, to fix a bug, add a new functionality, or improve the coding style. Authors submit their patches by email to *maintainers*, who commit the changes to their git trees and submit pull requests up the hierarchy. In this Chapter, we are most concerned with the maintainers, who have the responsibility of assessing the correctness and usefulness of the patches that they receive. Part of this responsibility involves determining whether a patch is stable-relevant, and annotating it accordingly.

The Linux kernel provides a number of guidelines to help maintainers determine whether a patch should be annotated for propagation to stable kernels. These are summarized as follows (slightly condensed for space reasons):[3]

- It must be obviously correct and tested.
- It cannot be bigger than 100 lines, with context.
- It must fix only one thing.
- It must fix a real bug that bothers people.
- It must fix a problem that causes a build error, an oops, a hang, data corruption, a real security issue, or some "oh, that's not good" issue.
- Serious issues as reported by a user of a distribution kernel may also be considered if they fix a notable performance or interactivity issue.
- New device IDs and quirks are also accepted.
- No "theoretical race condition" issues.
- It cannot contain any "trivial" fixes.
- It must follow the submittingpatches rules.
- It or an equivalent fix must already exist in Linus' tree (upstream).

These criteria may be simple, but are open to interpretation. For example, even the criterion about patch size, which seems completely unambiguous, is only satisfied by 93% of the patches applied to the stable versions based on Linux v3.0 to v4.7, as of April 16, 2017,[4] with other patches ranging up to 2754 change and context lines. More generally, different developers may have different strategies for choosing and propagating patches to stable kernels. Figure 5.1 shows the rate of propagation of patches from the various subsystems, where a subsystems is approximated as a subdirectory of `drivers` (device drivers), `arch` (architecture specific

[3] Documentation/process/stable-kernel-rules.rst

[4] Duplicates are possible, as a single patch may be applied to multiple stable versions.

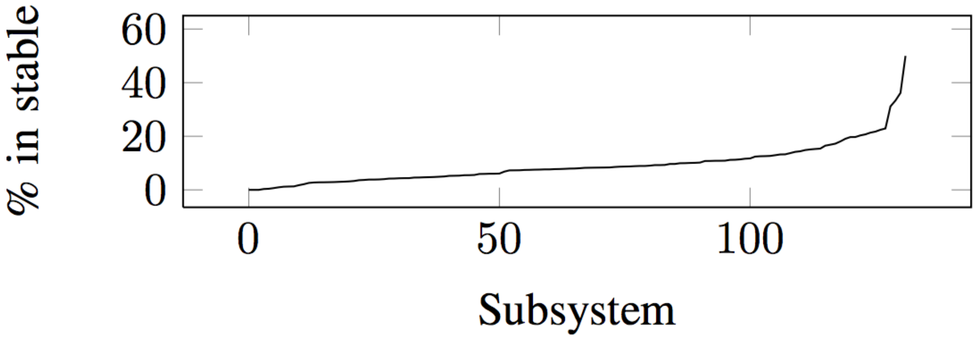


Figure 5.1: Rate at which the patches applied to a given subsystem end up in a stable kernel. Subsystems are ordered by increasing propagation rate.

support), or `fs` (file systems), or any other toplevel subdirectory of the Linux kernel source tree. While the median rate is 6%, for some subsystems the rate is much higher, raising the possibility that the median is too low and some stable-relevant patches are getting overlooked. If this is the case, part of the problem may be the stable propagation strategies of the individual maintainers. Indeed, as shown in Figure 5.2, the rate at which a given maintainer's commits that end up in a stable version are annotated with Cc stable covers the full range from 0-100%. While alternative submission options, *e.g.*, via email or via a pull request in the case of networking code, are listed in the stable kernel documentation, Cc: stable is advantageous because it is uniform and thus easy for developers to create tools against.

5.2.2 Challenges for Machine Learning

Stable patch identification poses some unique challenges for machine learning. These include the kind of information available in a Linux kernel patch and the diversity in the reasons why patches are or are not selected for application to stable kernels.

First, Linux kernel commit logs are written free-form. While maintainers are asked to add a `Cc: stable@vger.kernel.org` tag to commits that should be propagated to stable versions, our goal is to identify stable-relevant commits for which adding this tag has been overlooked, and thus we ignore this information.

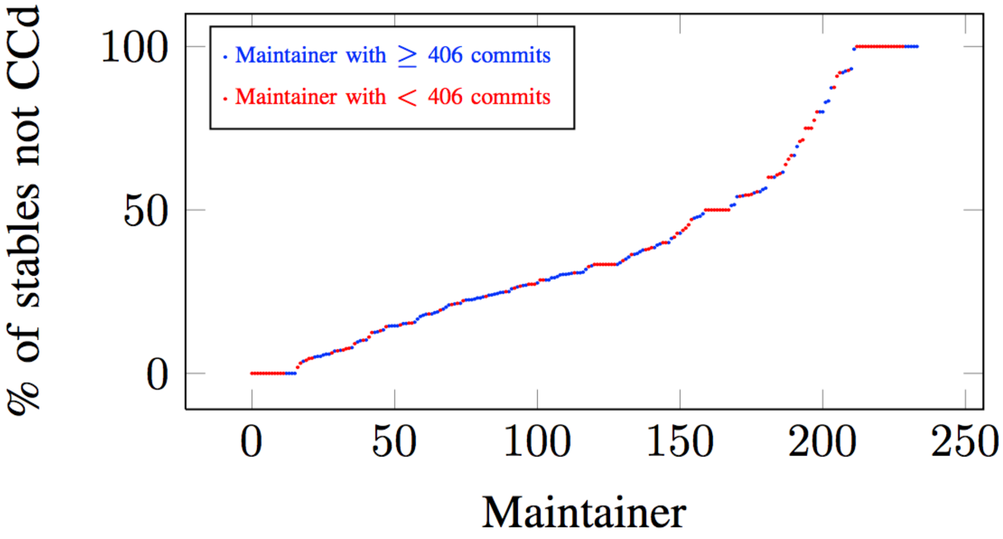


Figure 5.2: Rate at which a maintainer's commits that end up in a stable kernel are annotated with Cc stable. 406 is the median number of commits per maintainer. Maintainers are ordered by increasing Cc stable rate.

Patches also contain a combination of text, represented by the commit log, and code, represented by the enumeration of the changed lines. Code is structured differently than text, and thus we need to choose a representation that enables the machine learning algorithm to detect relevant properties.

Second, there are some patches that are applied to stable kernels that are not bug-fixing patches. The stable documentation itself stipulates that patches adding new device identifiers are also acceptable. Such patches represent a very simple form of new functionality, implemented as an array element initialization, but they are allowed in stable kernels because they are unlikely to cause problems for users of stable kernels and may enable the use of the stable kernel with new device variants. These patches have a common structure and are easily recognized, and thus should not pose a significant challenge for machine learning. Another reason that a non-bug fixing patch may be introduced into a stable kernel is that a subsequent bug fixing patch depends on it. These non-bug fixing patches, which typically perform refactorings, should satisfy the criteria of being small and obviously correct, but may have other properties that differ from those of bug-fixing patches. They may thus introduce apparent inconsistency into the machine learning process.

Finally, some patches may perform bug fixes, but may not be propagated to stable. One reason is that some parts of the code change so rapidly that the patch does not apply cleanly to any stable version. Another reason is that the bug was introduced since the most recent mainline release, and thus does not appear in any stable version.

As the decision of whether to apply a patch to a stable kernel depends in part on factors external to the patch itself, we cannot hope to achieve a perfect solution based on applying machine learning to patches alone. Still, stable-kernel maintainers have reported to us that they are able to check likely stable-relevant patches quickly (*e.g.*, 32 in around 20 minutes).[5] Therefore, we believe that we can effectively complement existing practice by orienting stable-kernel maintainers towards likely stable-relevant commits that they may have overlooked, even though the above issues introduce the risk of some false negatives and false positives.

5.2.3 Convolutional Neural Networks for Sentence Classification

In this Chapter, we leverage a convolutional neural network (CNN) to automatically learn features for stable related patch identification. In this section, we present background knowledge about how CNN is applied to automatically learn features for sentence classification task. We look at sentence classification tasks because the stable patch identification task could be modeled as a sentence classification task where each sentence contains all the information inside a patch.

In recent years, research and application of neural network based models have grown dramatically. Such models have achieved remarkable results in areas such as computer vision [38], speech recognition [22], and natural language processing (NLP) [67]. Many types of neural networks have been proposed, including Deep Belief Networks (DBN), Recurrent Neural Networks (RNN), Recursive Neu-

[5] Greg Kroah Hartman, private communication, April 28, 2017

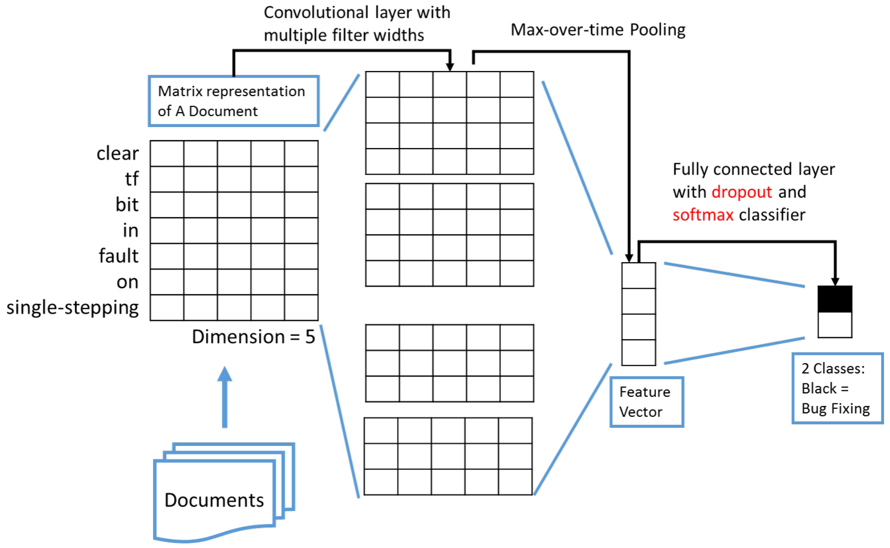


Figure 5.3: Convolutional Neural Networks for Sentence Classification

ral Networks (RNN), Convolutional Neural Networks (CNN), etc. In this work, we consider CNN, which utilize layers with convolving filters that are applied to local features [46]. CNN were originally designed for computer vision, and then have subsequently been shown to be effective for traditional NLP tasks, such as query retrieval [81], sentence modeling [32], and many more. Besides CNN's effectiveness in representing textual information, CNN are usually easier to train and have many fewer parameters than fully connected networks with the same number of hidden units.[6]

Figure 5.3 shows the architecture of the Convolutional Neural Network used in our approach. As shown in the figure, the input layer is a sentence (e.g, "clear tf bit in fault on single-stepping") comprised of concatenated word embeddings (i.e., representations of words using vectors). Each word is mapped to a vector of a fixed length (e.g., 5 in Figure 5.3). The input layer is followed by a convolutional layer with multiple filters, then a max-pooling layer, and finally a softmax classifier. This architecture is as the same as the CNN model proposed by Kim for sentence classification [36], except that it learns the word embeddings from the dataset itself rather than using any other pre-trained word vectors, based our hypothesis that word

[6] http://ufldl.stanford.edu/tutorial/supervised/ConvolutionalNeuralNetwork/

embeddings trained on any other domain or task might not be applicable to this software engineering specific task.

Convolution Layer & Max Pooling: Let $\mathbf{x}_i \in \mathbb{R}^k$ be the k-dimensional word vector corresponding to the i-th word in the sentence. A sentence of length n is represented as

$$\mathbf{x}_{1:n} = \mathbf{x}_1 \oplus \mathbf{x}_2 \oplus ... \oplus \mathbf{x}_n$$

where $\oplus$ is the concatenation operator. Sentences are padded such that they all have the same length as the maximum length sentence in the document. More generally, $\mathbf{x}_{i:i+j}$ refers to the concatenation of words $\mathbf{x}_i, \mathbf{x}_{i+1}, ..., \mathbf{x}_{i+j}$. A convolution operation involves a filter $w \in \mathbb{R}^{hk}$ that is applied to a window of h words to produce a new feature. For example, a feature c_i is generated from a window of words $\mathbf{x}_{i:i+h-1}$ by

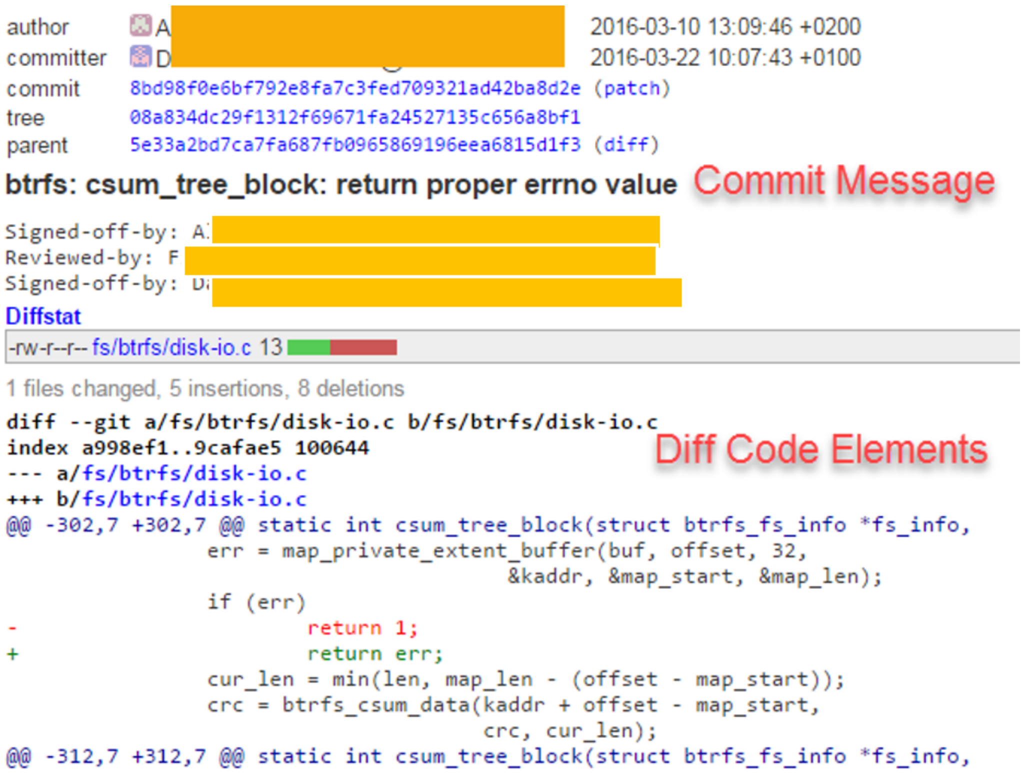
$$c_i = f(\mathbf{w} \cdot \mathbf{x}_{i:i+h-1} + b)$$

where $b \in \mathbb{R}$ is a bias term and f is a non-linear function, e.g., the hyperbolic tangent, i.e., $tanh$ in this CNN. This filter is applied to each possible window of words in the sentence $\{\mathbf{x}_{1:h}, \mathbf{x}_{2:h+2}, \ldots \mathbf{x}_{nh+1:n}\}$ to produce a feature map

$$\mathbf{c} = [c_1, c_2, ..., c_{n-h+1}]$$

with $c \in \mathbb{R}^{nh+1}$. Next, a max pooling operation [12] is applied over the feature map and takes the maximum value $\hat{c} = max\{c\}$ as the feature corresponding to this particular filter. The max pooling operation is designed to capture the most important feature, *i.e.*, the one with the highest value, for each feature map.

The above process describes how a feature is extracted from one filter. CNN uses multiple filters (with varying window sizes) to obtain multiple features. These features then form the penultimate layer and are passed to a fully connected softmax layer whose output is the probability distribution over two classes (i.e., stable-relevant and non stable-relevant fixing).



```
author     A[redacted]          2016-03-10 13:09:46 +0200
committer  D                    2016-03-22 10:07:43 +0100
commit     8bd98f0e6bf792e8fa7c3fed709321ad42ba8d2e (patch)
tree       08a834dc29f1312f69671fa24527135c656a8bf1
parent     5e33a2bd7ca7fa687fb0965869196eea6815d1f3 (diff)
```

btrfs: csum_tree_block: return proper errno value Commit Message

```
Signed-off-by: A[redacted]
Reviewed-by: F[redacted]
Signed-off-by: D[redacted]
```

Diffstat

-rw-r--r-- fs/btrfs/disk-io.c 13 🟩🟥

1 files changed, 5 insertions, 8 deletions

Diff Code Elements

```
diff --git a/fs/btrfs/disk-io.c b/fs/btrfs/disk-io.c
index a998ef1..9cafae5 100644
--- a/fs/btrfs/disk-io.c
+++ b/fs/btrfs/disk-io.c
@@ -302,7 +302,7 @@ static int csum_tree_block(struct btrfs_fs_info *fs_info,
                err = map_private_extent_buffer(buf, offset, 32,
                                        &kaddr, &map_start, &map_len);
                if (err)
-                       return 1;
+                       return err;
                cur_len = min(len, map_len - (offset - map_start));
                crc = btrfs_csum_data(kaddr + offset - map_start,
                                      crc, cur_len);
@@ -312,7 +312,7 @@ static int csum_tree_block(struct btrfs_fs_info *fs_info,
```

Figure 5.4: Sample Bug Fixing Patch

Regularization: A critical problem when applying machine learning algorithms is how to make algorithms consistently perform well on both training data and new data (testing data). Many methods have been proposed to modify algorithms to reduce their generalization error but not their training error. These methods are known collectively as regularization. The CNN considered in this Chapter adopts a regularization method called dropout, to prevent overfitting the learned neural network to the training data. A dropout layer stochastically disables a fraction of its neurons, which prevents neurons from co-adapting and forces them to individually learn useful features [25].

5.3 Approach

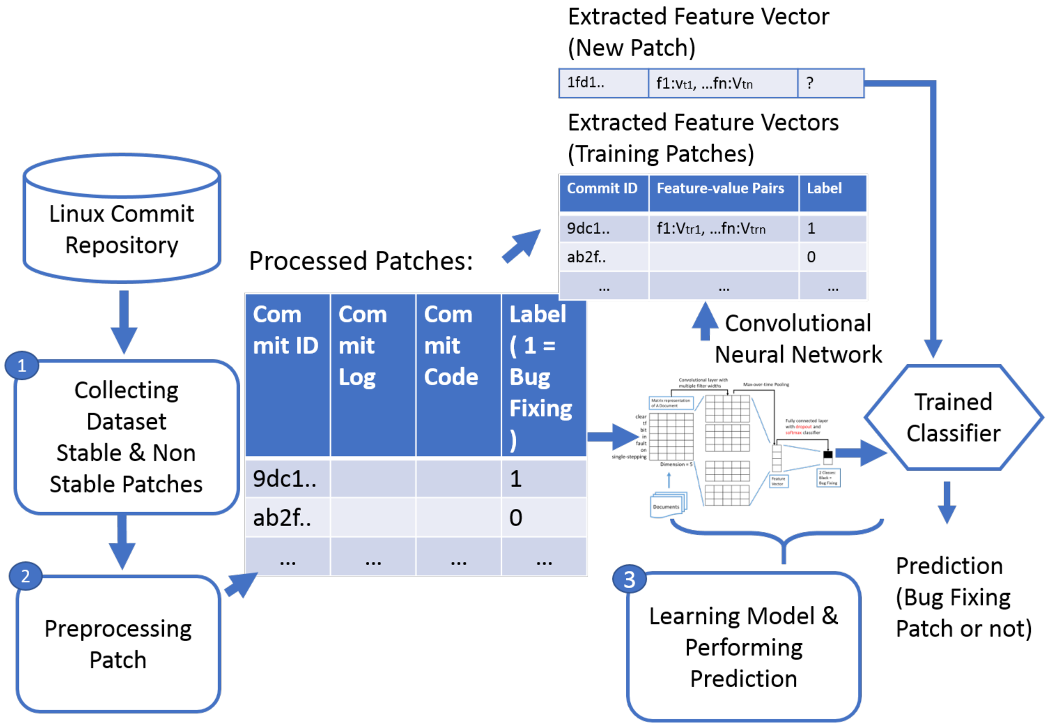
In this work, we leverage the model introduced in Section 5.2.3 to generate features and learn a classification model from the training patches. Figure 5.4 shows a sample patch that has been applied to the stable version derived from Linux v4.5

as the patch fixes user-visible bugs and improves error handling and stability. As illustrated in Figure 5.4, a patch contains not only a textual commit message but also a set of diff code elements, i.e., changes that are applied on the buggy file.

Figure 5.5 illustrates the framework of our approach. Our approach is composed of three steps: collecting the dataset, processing patches, and learning the model & performing prediction. In the first step, we collect a set of recent stable and non-stable patches from the commit repository of the Linux kernel, and annotate them as the stable-relevant patches (i.e., positive instances) and the non stable-relevant patches (i.e., negative instances), respectively. In the second step, we transform each collected patch into a document that a CNN could take as input. Note that as code is different from natural language, we process the commit message and the diff code elements separately and merge them into one document (see "Processed Patches" in Figure 5.5). In the last step, features/representations of training patches as well as a classifier (based on the learned features) are trained. The trained Convolutional Neural Network is able to convert any new patch into a set of feature-value vectors (see "Extracted Feature Vector (New Patch)" in Figure 5.5) and make a prediction by applying the trained classifier. We elaborate the details of each step in the rest of this section.

5.3.1 Collecting the Data Set

Similar to work presented in Chapter 4, we need to collect a set of stable patches (bug-fixing patches) as well as a set of non stable patches. For each patch, we collect the following information: commit id, author name, date on which the author provided the patch, committer name, date on which the committer committed the patch, the subject line, back link information (stable patches only), number of lines of changed code and context code (by default, typically the three lines before and after each *hunk* of contiguous removed and added lines). A back link is the reference to the same patch in the mainline Linux kernel. Next, we describe in more detail


Linux Commit Repository
1
Collecting Dataset Stable & Non Stable Patches
2
Preprocessing Patch
Processed Patches:
Extracted Feature Vector (New Patch)
Extracted Feature Vectors (Training Patches)
Convolutional Neural Network
Trained Classifier
3
Learning Model & Performing Prediction
Prediction (Bug Fixing Patch or not)

1fd1..	f1:Vt1, …fn:Vtn	?

Commit ID	Feature-value Pairs	Label
9dc1..	f1:Vtr1, …fn:Vtrn	1
ab2f..		0
…	…	…

Commit ID	Commit Log	Commit Code	Label (1 = Bug Fixing)
9dc1..			1
ab2f..			0
…	…	…	…

Figure 5.5: Framework of Convolutional Neural Network Based Stable-Relevant Patch Identification

how we collect the data set.

1) Collecting Stable Patches: The main challenge in the processing of stable patches is to link them to the corresponding patches in the mainline. Indeed, all patches accepted into a stable version must have previously been applied in the mainline. Some stable patches contain an explicit link back to the corresponding mainline commit. For others, we rely on the author name and the subject line. Subject lines typically contain information about both the change made and the name of the file or directory in which the change is made, and thus should be relatively unique. Accordingly, the collection of stable patches also collects a mapping of back link information such as a fixes tag to the commit identifier that contains the link and a mapping of a pair of an author email address and a patch subject to a commit identifier.

Following the stable patch rules, presented in Section 5.2.1, we keep only stable patches having at most 100 lines of changed and context code. Discarding the larger patches means that our approach may not be able to learn to recognize them, but we consider that such patches are anomalous, and treat special situations that may not generalize.

2) Collecting Mainline Patches: The collection of mainline patches is essentially the same as the collection of stable patches, except that no back links are collected. As for the stable patches, we limit mainline patches to those of at most 100 lines, to prevent the CNN from creating a model based only on patch size. A mainline patch is recognized as being a stable patch if a stable patch has the same author and subject as the mainline patch or if there is a back link in the stable patch to the mainline patch, according to the mappings collected during the stable patch collection process.

3) Constructing the training and testing datasets: From the set of mainline patches, we collect three sets of patches: 1) the complete set of stable patches, 2) a set of patches that are not recognized as being in any stable version, referred to as *non-stable*, to be used for training, and 3) a set of non-stable patches that are to be used for testing. For the first set, we use the complete set of stable patches for training, to have the most possible information to learn from. As our initial experiments with CNN showed that training works best when the data is balanced, we then extract the same number of non-stable patches for the second set as there are stable patches available for the first set. Finally, as our motivation is to help stable maintainers identify stable-relevant patches that would otherwise be overlooked, our testing data contains only non-stable patches as well. In addition to fitting with our objectives, this strategy has the benefit of allowing us to use the entire set of stable patches in the training data. We take a statistically significant subset of the set of non-stable patches of size at most 100 lines of changed and context code.

5.3.2 Patch Preprocessing

In this step, our approach takes patches collected in the first step as input and processes each one into a document. Each document contains a sequence of tokens that represent the patch. As mentioned before, our approach treats code changes separately from the commit log and combines them in the end to form a document. We describe in detail the methodology below.

1) Extract Atomic Statement Level Difference: Diff code elements may have many shapes and sizes - a single word, part of a line, an entire line, multiple lines, multiple lines separated by unchanged code, etc. To describe changes in terms of meaningful syntactic units and in particular to provide some context for very small changes, we collect differences at roughly the granularity of atomic statements. These may be, *e.g.*, simple assignment statements, but also if headers, for-loop headers, function headers, etc. We also distinguish changes in error checking code (code to detect whether an error has occurred) and in error handling code (code to clean up after an error has occurred) from changes in other code. Error checking code and error handling code are indeed very common in the Linux kernel, which must be robust, but are disjoint in structure and purpose from the implementation of the main functionality.

For a given commit, the first step is to extract the names of the affected files and to extract the state of those files before and after the commit. For each before and after file instance, we remove comments (taking care to preserve line numbers) and the contents of strings, as changes in comments and within strings are not likely to be stable-relevant. For a given pair of before and after files, we then compute the difference using the command "git diff -U0 old new". For each $-$ or $+$ line in the diff output, we then collect a record indicating the sign, the hunk number, the line number in the old or new version, respectively, and the starting and ending columns of the non-space changes on the line.

The previous step gives the differences, but the granularity may be below that of our atomic statements. For example, if a function call extends over multiple lines, the change could be in a single argument, on a line by itself. We thus then work on the old or new file individually, to map the changed lines to their enclosing atomic statements, as defined above. This process is performed using Coccinelle [63]. It is limited to the set of patterns supported by the Coccinelle script, and fails, causing the patch to be ignored, if there is any changed token that is not taken into account by these patterns or if Coccinelle is not able to parse the code.

As an example of the processing of code changes, consider the code snippets shown in Figure 5.6. In the before code, the `if` test expression is found to intersect with a changed line, so part of the result is the information about the if header, i.e., `if (x < 0)`. The `return` statement is also found to intersect with a changed line, so another part of the result is `return -1;`. Similar information is obtained for the after code. Due to the return in the if branch, the changed if headers are annotated as coming from error checking code, and the changed return statements are annotated as coming from error handling code. All of these changes are additionally annotated as coming from the same hunk.

```
Before:         After:
if (x<0)        if (y<0)
return -1;      return -2;
```

Figure 5.6: Code Example

2) Combining Statement Differences into a Code Representation: As a result of this phase, each hunk is represented as a sequence of tokens for the removed atomic statements followed by a sequence of tokens for the added atomic statements, at most one of which can be empty. We could simply concatenate these. To obtain a more precise view of the changes, we instead compute a word-level diff of the two sequences, using the command `git diff --word-diff=porcelain`, where the option porcelain produces the result in a format that eases subsequent processing. The result is a sequence of context (unchanged) tokens, intersprinkled with word-level hunks containing sequences of removed and added tokens. Rather than using word diff, we could alternatively have used tree differencing [20] to obtain fine-grained differences that would respect the programming language's syntactic structure. Word diff, however, is faster than tree differencing, because there is no need for parsing the source code, and thus we use word diff in the current approach.

In the result, we could treat the tokens in the diff code elements of a patch like words in the commit message. For example, Figure 5.4 could be treated as a document: "—a/fs/btrfs/disk-io.c +++ b/fs/btrfs/disk-io.c @@ -302,7..." However, developers may chose unique identifiers, such as "db1200_mmc_led", "db1200_mmc0_dev", "au1200_lcd_res", "au1200_lcd_dev", across different files and functions, even when the identifiers act similarly in the source code. Thus, if we consider all the tokens appearing in the diff code elements, the vocabulary size will be very large. The data will also be very sparse, because these identifiers might appear very few times across the data set. Thus, the extra information will provide little benefit for the learning process. To address this issue, the preprocessing of a patch ultimately drops the specific names of all identifiers, instead representing them all as a single "Ident" token.

3) Combine Code Representation with Commit Log: For each processed patch, we then combine the processed commit message and the diff code elements into a one-line document (with the symbol "##" as the line separator) so that the Convolutional Neural Network introduced in Section 5.2.3 can process it. We then use the *VocabularyProcessor* object from TFLearn [7] to map documents to sequences of numbers, where each number represents one word.

5.3.3 Learning Model & Performing Identification

In the last step, documents preprocessed from the training commits are used as input for training a convolutional neural network (CNN). The structure of this CNN is as the same as the one introduced in Section 5.2.3. As a CNN contains many input parameters, such as the number of filters, size of a filter, etc., we further split out a part of the data from the training data set to form a validation set. When we train

[7] A deep learning library featuring a higher level API for Tensorflow http://tflearn.org/.

a model from the rest of training data, we periodically test the performance (i.e., F-measure of all classes) of the latest model on the validation set. We stop training once the performance on the validation data starts to degrade, suggesting overfitting of the data. We then use the saved trained model to predict labels for unseen patches.

5.4 Evaluation

In this section, we first describe the dataset that is collected for the evaluation. Next, we describe the experimental settings for our CNN-based approach and the baseline approach. In the end, we describe our evaluation methodology and evaluation metric.

5.4.1 Dataset

To evaluate our approach, we target mainline versions 3.0 to 4.7. The stable versions build on mainline versions 3.0 to 4.6.[8] Given the fact that there are many more non stable patches existing in the mainline, for the training and evaluation purpose, we randomly collect a significant sample set from all non stable patches. For training, we have collected 16,265 stable patches, and a randomly sampled 14,688 non-stable patches. For evaluation, we again collected another randomly sampled 17,967 non-stable patches. In total, we considered 48,920 patches from Linux. All selected patches have at most 100 lines of change and context code, as stipulated in the stable kernel rules (see Section 5.2.1).

5.4.2 Model Settings

From all the training patches, we use 90% of them for training the CNN, while the remaining 10% are used to tune input parameters for the CNN. Our experiment code is written in Python and built on top of the TensorFlow Python library [1]. A

[8] The stable version building on *e.g.*, mainline version 4.6 is maintained in parallel with the preparation for version 4.7, and potentially onward.

Convolutional Neural Network contains multiple parameters. In this experiment, we choose to embed each token in the document that is processed from a patch into a vector of length 16. We consider filters of two different window sizes, i.e., 4 and 5. The number of filters is set to 32. During training, the dropout ratio is set to 0.5, which means that 50% of the units will be dropped out randomly during training. The values of these input parameters were chosen in two steps. We first considered values that were found to be good in prior studies [27, 36]. We then heuristically tuned the input parameters based on the performance of the corresponding models on the validation set.

5.4.3 Baseline Approach

We take the LPU+SVM based approach proposed in Chapter 4 as the baseline approach. The LPU+SVM based approach extracts features from both code changes and commit logs that can potentially distinguish bug fixing patches from regular commits. We predefined the features. The input to the LPU+SVM based approach is a set of bug fixing patches (for stable versions) and unlabeled patches. In our experiments, we currently treat the non stable patches as the unlabeled patches for the baseline approach.

5.4.4 Evaluation Methodology & Metrics

Evaluation Methodology: We evaluate the performance of the two stable patch identification approaches on the testing data, which is a significant sample set that is randomly selected from the non-stable patches. To avoid the bias that may be introduced by self labeling, we ask the Linux kernel stable version maintainers, Greg Kroah-Hartman and Sasha Levin, to help us evaluate the results. To save their time in labeling, we prepare two datasets for them to label:

Option 1 For each approach, we rank all the testing patches (currently not in the stable tree) based on their probability of being stable, i.e., top patches are considered by the classifier to be most likely to be stable. We then take the top-50 patches for each approach and create a set that covers all these patches. We sent these patches to Greg Kroah-Hartman to label.

Option 2 We randomly select 100 patches that are currently not applied to any considered stable version and send them to Sasha Levin to label.

Note that to avoid bias, we told both developers that the patches were randomly selected from those not in the stable versions. Thus option 1 and option 2 are treated equally.

Evaluation Metrics: Our goal in evaluation option 1 is to compare the ground truth labels provided by the maintainers with the ranking of the patches within the top 50 results produced by each classifier. We thus use two common ranking-based evaluation metrics that evaluate the quality of a ranked list:

- **Accuracy@N**: this metric calculates the ratio of real stable patches in the top-N list provided by each approach.

- **Average precision (AP)@N**: The average precision of a ranked list of potential stable patches is computed as:

  $$AP = \sum_{k=1}^{M} \frac{P(k) \times pos(k)}{\textit{number of stable patches}}$$

  where k is a rank in the returned ranked patches, M is the number of ranked patches, and $pos(k)$ indicates whether the kth patch is stable or not. $P(k)$ is the precision at a given top k and is computed as follows:

  $$P(k) = \frac{\textit{\#stable patches top k}}{k}$$

- **Normalized Discounted Cumulative Gain (NDCG)@N** NDCG measures the performance of a recommendation system based on the graded relevance of the recommended entities. It varies from 0.0 to 1.0, with 1.0 representing the ideal ranking of the entities. This metric is commonly used in information retrieval and to evaluate the performance of web search engines.

For evaluation option 2, we calculate commonly used evaluation metrics for classification tasks, i.e., precision, recall, and F-measure, for both approaches.

5.5 Evaluation Results & Discussion

5.5.1 CNN-based Approach vs. LPU+SVM based Approach

Evaluation option 1: Applying the CNN-model and the baseline to the testing data, which are all non-stable patches, the CNN-model identified 4,710 (26.2%) of them to be stable, while the baseline identified 4,341 (24.2%) of them to be stable. After we rank all the testing non stable patches according to their probability of being stable for both approaches, we find that there is only one commit that overlaps between the top-50 lists of the two approaches.[9] With his labels, Greg Kroah-Hartman noted "Overall, very nice work in finding lots of patches that are relevant."[10]

Table 5.1 shows the Accuracy@N and AP@N of the CNN-based approach and the LPU+SVM based approach. These metrics show that both approaches can capture missing stable patches if we consider the top-50 patches as the most likely stable ones. And in most of the cases (2/5 for Accuracy@N, 4/5 for Average Precision@N, and all NDCG@N), the CNN-based approach performs better than the LPU+SVM based approach, which means the probability returned by the CNN-based approach could help generate a better ranking of the top-50 results. However we also notice that the difference between two approaches is minor.

[9] 5c2e08231b68a3c8082716a7ed4e972dde406e4a

[10] Private communication, May 12, 2017.

Evaluation option 2: The precision, recall and F-measure of the CNN-based approach and the LPU+SVM approach are shown in Table 5.2. We notice that the potential of capturing stable patches (i.e., recall) are equal for the two approaches. On the other hand, the CNN-based approach is more aggressive in predicting a patch as a stable one, and thus is has a lower (i.e., 7.7%) precision than the LPU+SVM based approach. Similar to the results based on the first evaluation option, the difference between two approaches is minor.

Table 5.1: Accuracy@N, Average Precision (AP)@N, Normalized Discounted Cumulative Gain (NDCG)@N: CNN vs. LPU+SVM

Approach	**Acc@10**	**Acc@20**	**Acc@30**	**Acc@40**	**Acc@50**
LPU+SVM	0.9	0.85	0.9	0.9	0.9
CNN-based	1	0.9	0.83	0.825	0.84
Difference	**+11%**	**+5.9%**	-7.8%	-8.3%	-6.7%

	AP@10	**AP@20**	**AP@30**	**AP@40**	**AP@50**
LPU+SVM	0.976	0.925	0.909	0.909	0.909
CNN-based	1	0.979	0.948	0.922	0.905
Difference	**+2.4%**	**+5.8%**	**+4.3%**	**+1.4%**	-0.4%

	NDCG@10	**NDCG@20**	**NDCG@30**	**NDCG@40**	**NDCG@50**
LPU+SVM	0.99	0.98	0.975	0.975	0.975
CNN-based	1	0.995	0.988	0.982	0.979
Difference	**+1%**	**+1.5%**	**+1.3%**	**+0.7%**	**+0.4%**

Table 5.2: Precision, Recall, F-measure: CNN vs. LPU+SVM

Approach	**Recall**	**Precision**	**F-measure**
LPU+SVM based	0.545	0.75	0.631
CNN-based	0.545	0.692	0.610
Difference	0%	-7.7%	-3.3%

5.5.2 Potential of Combining the CNN-based and LPU+SVM-based Approaches

By checking the top-50 patches returned by the two approaches for evaluation option 1, we find that although most of the patches in the top-50 are stable patches, out of the total 100 patches, 99 of them are unique. Thus, there maybe some complementarity between the approaches We then investigate the results obtained by the two approaches in further detail, to gain insight into the potential of combining them.

We evaluate the agreement of two approaches in two ways: 1) by considering the ranked lists provided by the two approaches on the whole testing dataset; 2) by comparing their classification results on the 199 manual labeled patches (see Section 5.4.4 for the creation of this set of 199 patches).

Correlation between Ranked Lists: We test the correlation between two ranked lists provided by the CNN-based approach and the LPU+SVM based approach using Kendall's τ coefficient [2]. The input for calculating the Kendall τ coefficient is (x_1, y_1), (x_2, y_2),..., (x_n, y_n) - a set of observations of the joint random variables X and Y respectively. All the values of x_i and y_i are unique. The Kendall τ coefficient is defined as:

$$\tau = \frac{(\textit{number of concordant pairs}) - (\textit{number of discordant pairs})}{n(n-1)/2}$$

For any pair of observations (x_i, y_i) and (x_j, y_j), where $i \neq j$, are said to be concordant if the ranks for both elements agree, e.g., if both $x_i > x_j$ and $y_i > y_j$, or if both $x_i < x_j$ and $y_i < y_j$. The pair is discordant if $x_i > x_j$ and $y_i < y_j$, or if $x_i < x_j$ and $y_i > y_j$. If $x_i = x_j$ or $y_i = y_j$, the pair is neither concordant nor discordant.

Kendall's τ coefficient ranges between -1 and 1, with -1 indicating that the rankings are completely different, 0 indicating no correlation, and 1 indicating that the results are correlated.

We find that Kendall's τ coefficient between the ranked lists provided by CNN and that provide by LPU+SVM is 0.482, which indicates a moderate correlation between the two ranked lists. The moderately correlation shows consistency between the two approaches, i.e., they have similar judgments (probability of being stable) on many patches. However, there are inconsistencies on some cases as well, which might be further investigated for a potential combination of the two approaches.

Correlation between the Classification Results: In this setting, we focus on the prediction results (i.e., stable or not) of the CNN-based and the LPU+SVM based approach, instead of the detailed ranking position of each testing patch.

Table 5.3 shows the predictions of two approaches on the complete set of 17,967 testing patches. The results show that the CNN based approach predicted more patches as stable ones, with 26.2% of test patches are identified as stable, while the LPU+SVM based approach identified 24.2% of test patches as stable. We also find that many (65.6%) patches are identified as non stable patches by both approaches. Beside the patches identified by both approaches as stable, there are 3,325 (i.e., 1,847 + 1,478) patches that are labeled as stable by one but not the other. If there are some missing stable patches in these 3,325 patches, then combining the two approaches might help find more missing stable patches.

Table 5.3: Predictions of CNN and LPU+SVM on 17,967 Testing Patches

(a) #Stable Patches identified by both Classifiers.	2,863
(b) #Stable Patches identified only by CNN	1,847
(c) #Stable Patches identified only by LPU+SVM	1,478
(d) #Non Stable Patches identified by both	11,779

Table 5.4 and Table 5.5 shows the performance of two approaches on the 199 unique labeled patches for two evaluation options. We find that although the top-50 patches overlap in only one case for the two approaches, still both approaches annotate many of the same top-50 patches as stable, with a different ranking. Indeed, Table 5.4 shows that out of the 99 unique patches in the two top-50 lists, 86 are actually stable and 79 of these 86 are predicted to stable by both approaches. But if we look at the performance of the two approaches on the randomly sampled 99 patches in Table 5.5, there is a potential benefit of combining two approach, to get 20 (i.e., 9 + 11) more correctly predicted patches.

Table 5.4: Performance of CNN and LPU+SVM on 99 Patches (Option 1)

(a) #Stable Patches found by both Classifiers.	79
(b) #Stable Patches found only by CNN	6
(c) #Stable Patches found only by LPU+SVM	1

Table 5.5: Performance of CNN and LPU+SVM on 100 Patches (Option 2)

(a) #Patches correctly identified by both	66
(b) #Patches correctly identified only by CNN	9
(c) #Patches correctly identified only by LPU+SVM	11
(d) #Patches wrongly identified by both	14

5.5.3 Threats to Validity

In Chapter 4, we manually checked the labels of 500 sampled patches to establish the ground truth, which might introduce experiment bias. Thus, in this work, we instead regard all the labels already been assigned by the Linux maintainers as the ground truth. However in reality, Linux maintainers might also make mistakes, particularly, they might miss some bug fixing patches. We plan to mitigate such threats by asking some Linux maintainers to label parts of the evaluation data set again in the future.

In Chapter 4, we randomly selected 500 commits and use them to evaluate our prediction model. This time, we ask the real practitioners to evaluate the two approaches. Similar to the evaluation in Chapter 4, we only investigated patches from the Linux project, although the CNN based approach can be easily applied to identify bug fixing patches in other systems. In the future, we would like to consider more projects.

The selection of evaluation metrics might introduce threats to construct validity. To mitigate such threats, we consider the standard measures, i.e., precision, recall, F-measure, and accuracy [54] to evaluate the effectiveness of a bug fixing patch identifier.

5.6 Chapter Conclusion

In this chapter, we have revisited the problem of identifying stable related patches (proposed in Chapter 4) with a new Convolutional Neural Networks (CNN) based approach. The new approach takes both the commit log and preprocessed patch code as input and automatically learns representations/features from the patch for better classification.

We collect a new set of stable and non stable patches from recent Linux mainline versions and stable versions. This new dataset contains 48,920 patches. To investigate the potential benefit of using CNN, we compare the performance of the CNN-based approach and the LPU+SVM based approach proposed in Chapter 4 by asking maintainers of Linux stable versions to label our results. Our results show that the CNN-based approach performs similar to the LPU+SVM approach, and it does not require any hand-crafted features. This comparison indicates that future customization of the current CNN-based approach might be considered by other researchers for better identification of stable related patches.

Chapter 6

Related Work

In this chapter, we introduce related studies on mining software repositories for bug management and applying deep learning techniques on software engineering tasks. We also describe how these studies are related to the work presented in this thesis.

6.1 Duplicate Bug Report Detection

A number of approaches have been proposed to detect duplicate bug reports. Many of these approaches rely on a good similarity measure to find bug reports that are close to one another. These include work by Runeson et al. [75], Wang et al. [105], Jalbert and Weimer [29], Sun et al. [85, 86], and many more.

These studies represent a bug report as a vector of feature values extracted from the various fields of the bug report. These vectors of feature values are then compared with one another and a similarity score is computed. All of these studies consider the textual description available in bug reports. Many of them make use of the concepts of term frequency and inverse document frequency to determine the importance of the word tokens appearing in bug reports. The work by Wang et al. [105] considers execution traces in addition to words in the bug reports; they have shown that execution traces, if present in bug reports, could be used to detect duplicate bug reports accurately. The work by Jalbert and Weimer [29] and Sun et al. [85] consider other non-textual fields in a bug report, e.g., the product that is impacted by the bug, etc., to measure the similarity of two bug reports.

Users of software systems may report bugs that are already present in the bug tracking system, since bug reporting is an uncoordinated and distributed process. These duplicates need to be manually labeled as such during the bug triage process, which takes considerable human effort and time. A number of automated duplicate bug report detection approaches have thus been proposed [75, 85, 86, 105]. Given a new bug report, these approaches return a list of previously reported bugs which are similar to the new report. Runeson et al. extract words from the bug report description and summary fields and use cosine, dice, and jaccard similarity to measure the similarity of reports [75]. Sun et al. consider not only text in bug reports, but also many other non-textual fields in the bug reports, e.g., product, etc., to capture degree of relevance between two bug reports [85, 86]. They propose a machine learning approach and extend a variant of BM25 to retrieve duplicate reports. Wang et al. enrich textual information from bug reports with execution traces to more accurately detect duplicate bug reports [105].

Relation to this thesis: In our automated bug prioritization approach (proposed in Chapter 2), we analyze multiple factors that might impact the priority level of a bug report, which include the priority levels of similar bugs, i.e., the related-report factor. For the related-report factor, we capture the mean and median priority of the top-k reports as measure using REP^{-}. REP^{-} is a bug report similarity measure adapted from the studies by Sun et al. [85] – described in Section 2.2.

6.2 Bug Severity and Priority Prediction

Menzies and Marcus were the first to predict the severity of bug reports [56]. They analyze the severity labels of various bugs reported in NASA. They propose a technique that analyzes the textual contents of bug reports and outputs *fine-grained* severity levels – one of the 5 severity labels used in NASA. Their approach extracts word tokens from the description of the bug reports. These word tokens are then pre-processed by removing stop words and performing stemming. Important word tokens are then selected based on their information gain. Top-k tokens are then used as features to characterize each bug report. The set of feature vectors from the training data is then fed into a classification algorithm named RIPPER [11]. RIPPER learns a set of rules that are then used to classify future bug reports with unknown severity labels.

Lamkanfi et al. extend the work by Menzies and Marcus to predict severity levels of reports in open source bug repositories [41]. Their technique predicts if a bug report is severe or not. Bugzilla has six severity labels including `blocker`, `critical`, `major`, `normal`, `minor`, and `trivial`. They drop bug reports belonging to the category `normal` because `normal` is the default severity level. The remaining five categories are grouped into two groups – severe and non-severe. The severe group includes `blocker`, `critical` and `major`. The non-severe group includes `minor` and `trivial`. Thus, they focus on the prediction of *coarse-grained* severity labels.

Extending their prior work, Lamkanfi et al. also try out various classification algorithms and investigate their effectiveness in predicting the severity of bug reports [42]. They tried a number of classifiers, including Naive Bayes, Naive Bayes Multinomial, 1-Nearest Neighbor, and SVM. They find that Naive Bayes Multinomial perform the best among the four algorithms on a dataset consisting of 29,204 bug reports.

Recently, Tian et al. also predict the severity of bug reports by utilizing a nearest neighbor approach to predict fine-grained bug report labels [97]. Different from the work by Menzies and Marcus which analyzes a collection of bug reports in NASA, Tian et al. apply the solution on a larger collection of bug reports consisting of more than 65,000 Bugzilla reports.

Khomh et al. automatically assign priorities to Firefox crash reports in Mozilla Socorro server based on the frequency and entropy of the crashes [34]. A crash report is automatically submitted to the Socorro server when Firefox fails and it contains a stack trace and information about the environment to help developers debug the crash.

Relation to this thesis: In Chapter 2, we proposed an automated bug prioritization approach for bug reports that are manually submitted by users. Different from a crash report, a bug report contains natural language descriptions of a bug and might not contain any stack trace or environment information. Thus, different from Khomh et al.s approach, we employ a text mining based solution to assign priorities to bug reports. Bug prioritization is orthogonal to the above studies on bug severity label prediction. Severity labels are reported by users, while priority levels are assigned by developers. Severity labels correspond to the impact of the bug on the software system as perceived by users while priority levels correspond to the importance “a developer places on fixing the bug” in the view of other bug reports that are received (Eclipse 2012).

6.3 Bug Report Assignee Recommendation

Studies on bug assignee recommendation can be categorized into two groups based on their underlying mechanism: activity-based [4, 60] and location-based approaches [26, 50, 82].

A number of activity-based bug assignee recommendation approaches have been presented in the literature. For example, Cubranic and Murphy collect features from the description and summary fields of bug report and build a Naive Bayes classifier for determining the similarity between the expertise of a developer and a new bug report [60]. Later, Anvik and Murphy compare the performance of various machine learning techniques for automatic bug report assignee recommendation task [4], and show that the Support Vector Machine (SVM) classifier performs the best among several commonly-used classifiers. Most of these approaches use term-weighting techniques, such as term frequency-inverse document frequency (tf-idf), to determine the value of word features.

Similarly, a number of location-based bug assignee recommendation approaches have been presented in the literature. For example, Linares-Vasquez et al. used Latent Semantic Indexing (LSI) to first locate potential source files related to a change request and then recommend developers using authorship information in the corresponding source files [50]. Later, Hossen et al. extend Linares-Vasquez et al.'s work by adding more information, i.e., maintainers of relevant source code and change proneness of source code [26]. Shokripour et al. propose a two-phase location-based approach to leverage multiple information sources including identifiers and comments in source code files, commit messages, and previous fixed bug reports [82].

Relation to this thesis: Both activity-based and location-based bug assignee recommendation approaches have advantages and disadvantages. In this thesis, we combine the two to build a unified bug assignee recommendation model. This model is presented in Chapter 3. Our experimental results also show that our unified model performs the best when compared to a location-based baseline by Anvik et al. [4] and an activity-based baseline by Shokripour et al. [82].

6.4 IR-based Bug Localization

Bug localization, which locates source files potentially responsible for the bugs reported in bug reports, is an important but costly activity in software maintenance. Most existing approaches treat the source files as documents and formalize the bug localization problem as a document retrieval problem. Various models have been constructed to compute the similarity or relevancy between the bug reports and the source files. Many information retrieval based bug localization methods have been proposed [21, 52, 69, 111, 112].

Poshyvanyk et al. propose a feature location model to mine buggy files based on a Latent Semantic Indexing (LSI) model, which can identify the relationship between reports and terms based on Singular Value Decomposition (SVD) [69].

Lukins et al. apply a generative probabilistic model, i.e., the Latent Dirichlet Allocation (LDA) model, to model how source code files are generated from words through topics [52]. Given a bug report, their approach applies the LDA model learned from source code to calculate the probability of generating the bug report from a source code file. The code files that are more likely to generate the bug report are returned as the buggy files. Gay et al. [21] employ the Vector Space Model (VSM) based on concept localization to represent bug reports and source code files as feature vectors, which are used to measure the similarity between bug reports and source files. Zhou et al. [112] propose BugLocator using a revised Vector Space Model (rVSM), which is based on document length and similar bugs that have been resolved before as new features. Lam et al. [40] employ auto encoder to learn features that correlate the frequently occurred terms in bug reports and source files in order to enhance the bag-of-words features. Recently, Ye et al. propose an approach that combines multiple ranking features leveraging learning-to-rank technique [111]. These features include surface lexical similarity, API-enriched lexical similarity, collaborative filtering, class name similarity, etc.

Concern localization is another line of work that is related to IR-based bug localization [18, 45, 103]. Many concern localization approaches could be applied for locating bug reports, as their fundamental assumption is the same as that of IR-based bug localization, i.e., treating concern localization as an information retrieval task.

Relation to this thesis: In our automated bug assignment approach (proposed in Chapter 3), we consider four location-based features. These features are calculated based on the output of a bug localization approach. And we take the latest approach proposed by Ye at al. [111]. Similar to Ye et al., our automated bug assignment approach also adopts the learning-to-rank technique, as it could nicely combine various metrics for measuring the similarity between a bug and a developer.

6.5 Identification of Bug Fixing Patches

Bird et al. have observed that the lack of clearly identified bug fixing patches itself has caused potential bias in many prior studies [6]. A number of studies have searched for keywords such as "bug" and "fix" in log messages to identify bug fixing commits [16, 35, 57, 83].

There are two other studies that are related to bug fixing patches identification. Wu et al. propose ReLink which links bug reports to their associated commits [108]. ReLink only captures tracked bugs; bugs described only in mailing lists, etc. are mentioned as future work. Our work considers a different problem and does not require the availability of bug reports, which may be absent or incomplete. Bird et al. propose Linkster which integrates information from various sources to support manual link recovery [7].

Relate to this thesis: In Chapter 4, we mention that keyword based approaches are not sufficient because not all bug fixing commit messages include the pre-defined keywords. Our approach in Chapter 4 addresses such limitation by automatically inferring keywords that are good at discriminating bug fixing patches from other commits. Furthermore, we consider not only commit logs, but also some features extracted from the changes made to the source code. We then built a discriminative machine learning model (i.e., LPU+SVM based approach) that is used to classify commits as either bug fixing or not. In Chapter 5, we propose another convolutional neural network (CNN) based approach to identify bug fixing patches that should be considered in the Linux stable versions. Beside the new approach, we invited real practitioners to evaluate the performance of both our CNN-based and LPU+SVM based approaches. Such evaluation setting is never considered in any prior related studies.

6.6 Deep Learning in Software Engineering

Deep learning, as a powerful representation learning technique, after finding success in Natural Language Processing (NLP) and Computer Vision (CV) field, has been recently applied to solve software engineering tasks. These tasks including defect prediction [101], bug localization [33], program representation learning [66, 59], summarizing code using natural language [28], program synthesis [49], program inductions [23, 39]. In these applications, the authors either leverage standard types of neural networks or design their own neural networks for specific tasks.

Relation to this thesis: Similar to the above approaches, we leverage a convolutional neural network to learn a better representation of a patch that could contribute to the identification of bug fixing patch in Chapter 5. Such an application has not been done by any prior work.

Chapter 7

Conclusion and Future Work

7.1 Conclusion and Contributions

Due to system complexity and inadequate testing, modern software systems are often released with bugs. The bug resolving process plays an important role in the development and evolution of software systems, so as to improve the quality of software systems until the next release. Every day developers could collect a considerable number of bugs from users and testers. To help developers effectively address and manage these bugs, bug tracking systems such as Bugzilla and JIRA are adopted to manage the life cycle of a bug through bug report. These bug repositories, and their linked code corpus, contain a wealth of valuable information. Such information could be mined to automate bug management process and thus save developers time and effort.

This thesis focuses on two stages in the life of a bug, i.e., the bug triaging stage before developer starts fixing a bug, and the patch backporting stage after a bug has been fixed by a developer through a patch. We aim to automate two specific tasks happen in the bug triaging stage, i.e., bug prioritization and bug assignment. Detailed proposed approaches were presented in Chapters 2 and 3, respectively. For the patch backporting stage, we aim to automate one task, i.e., bug fixing patch identification for stable versions. We propose two approaches, one is based on

LPU (Learning from Positive and Unlabeled Examples) +SVM (Support Vector Machine) and hand-crafted features, and the other is based on Convolutional Neural Networks (CNN). The corresponding approaches are presented in Chapter 4 and 5, respectively.

The contributions of this thesis are:

1. We propose the new problem of predicting the priority of a bug given its report. Past studies on bug report analysis have only considered the problem of predicting the severity of bug reports, which is an orthogonal problem.

2. We predict priority by proposing a new machine learning framework, named DRONE. DRONE considers various factors (i.e., `temporal`, `textual`, `author`, `related-report`, `severity`, and `product`) that potentially affect the priority level of a bug report. DRONE also contains a new classification engine, named GRAY, that *enhances* linear regression with *thresholding* to handle imbalanced data. We have experimented with our solution on more than a hundred thousand bug reports from Eclipse to evaluate its ability to support developers in assigning priority levels to bug reports. The results show that DRONE can outperform a baseline approach, built by adapting a bug report severity prediction algorithm, in terms of average F-measure, by a relative improvement of up to 209%.

3. We propose a unified model to predict the assignee of a new bug report based on the learning to rank machine learning algorithm. This unified model leverages information from both developers' activities and the result of bug report localization, thus the model integrates activity-based and location-based bug assignee recommendation approaches. Experimental results on more than 11,000 bugs from three open source projects show that combining location-based features and activity-based features through the learning to rank technique can improve the performance over using only one type of feature.

4. We identify the new problem of finding bug fixing patches to be integrated into a Linux “longterm” release.

5. We propose a LPU+SVM based approach to identifying bug fixing patches leveraging both textual and hand-crafted code features. The approach is based on two machine learning techniques, LPU and SVM. We combine these techniques to address the problem of unavailability of a clean negative dataset (i.e., non bug fixing patches). We have evaluated our approach on commits in Linux and show that our approach can improve on the keyword-based approach by up to 45.11% recall while maintaining similar precision.

6. We propose a Convolutional Neural Network (CNN) based approach to automatically identify patches for stable versions. The CNN-based approach does not require hand-crafted features. We compare the performance of the CNN-based approach with the LPU+SVM based approach on a larger set of recent Linux patches. The evaluation is under the help of maintainers of Linux stable versions. Our experimental results show that new CNN-based approach achieves a similar performance compared to the LPU+SVM based approach.

7.2 Future Work

7.2.1 As Completion of Previous Studies

Bug Prioritization

Our approach (see Chapter 2) might suffer from the cold start problem when applied on new/small projects because it might be hard to collect enough training data for learning an effective prediction model. To mitigate this cold start problem, I would like to apply transfer learning techniques, which allow the domains, tasks, and distributions used in training and testing to be different, thus general features/knowledge could be learned from large project and contribute to prediction on new/small projects. A similar idea has been applied to predict defects among multiple projects [109].

Bug Assignment

Our approach (see Chapter 3) takes output from the bug localization task as input for extracting location-based features. However, bug localization techniques keep evolving and their performance might impact the result of our unified model. Thus, as a future work, I would like to investigate impact of bug localization technique on our model and possibility of integrating code authorship and bug localization information directly in the bug assignment model.

Identification of Stable Release Related Patches

We have tried both hand-crafted features with a modified classifier (LPU+SVM, see Chapter 4) and a Convolutional Neural Network (CNN) based approach (see Chapter 5) to retrieve patches for Linux stable versions. Our results show the potential of catching missing stable patches by both approaches, based on our evaluation with real practitioners. We also find that the CNN-based approach could not improve on the performance of the LPU+SVM based approach. In the future, we plan to design a more complex representation for a patch. We plan to start by asking practitioners questions such as “what do you look for when manually identifying stable related patches?”.

7.2.2 Others

A Joint Model for Automated Bug Management

In this thesis, we have automated three bug management tasks individually. However, in practice, some of the bug management tasks could be considered together. For instance, a more general goal is, given a bug report with some fields are known, automatically recommend values for the other fields for facilitating developers in reproducing/fixing the bug. To achieve this goal, a joint model for bug field value generating process could be learned from historical bug reports. Ideally, this joint model could simultaneously provide predictions/recommendations to multiple labels of a bug, such as duplicate, assignee, and priority, given various information, such as the description of the bug and other available information, when the bug is reported.

Automated Bug Management Outside Bug Tracking System

In this thesis, we mine information mostly from bug tracking systems. However, many bug management processes do not exist in bug tracking systems. For instance, many Github projects encourage contribution of a bug-fix using GitHub's Pull Request work flow. In other case, discussion about bugs maybe stored in the archives of mailing lists. In such cases, how to better manage bugs is a new challenge facing developers, especially open source project developers Thus, we are considering how to adapt our automation techniques to other types of bug repositories besides traditional bug tracking systems.

Mining Bug Fixing Behavior

In this thesis, we mine software repositories to automate software engineering tasks. But mining software repositories, especially bug tracking system, could also improve software quality by supporting developers in the bug fixing process. Such support for bug fixing could have various forms, such as summarizing the bug fixing process given a resolved bug. The summarization could include the linkage between the bug symptom and the bug cause, as well as the linkage between the bug cause and the bug fix pattern.

Bibliography

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] H. Abdi. The kendall rank correlation coefficient. *Encyclopedia of Measurement and Statistics. Sage, Thousand Oaks, CA*, pages 508–510, 2007.

[3] P. Achananuparp, I. N. Lubis, Y. Tian, D. Lo, and E.-P. Lim. Observatory of trends in software related microblogs. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 334–337. ACM, 2012.

[4] J. Anvik and G. C. Murphy. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(3):10, 2011.

[5] A. Bachmann and A. Bernstein. Software process data quality and characteristics: a historical view on open and closed source projects. In *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, pages 119–128, 2009.

[6] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: bias in bug-fix datasets. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 121–130. ACM, 2009.

[7] C. Bird, A. Bachmann, F. Rahman, and A. Bernstein. Linkster: enabling efficient manual inspection and annotation of mined data. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 369–370. ACM, 2010.

[8] C. Cadar, D. Dunbar, D. R. Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[9] Y.-W. Chang and C.-J. Lin. Feature ranking using linear svm. In *WCCI Causation and Prediction Challenge*, pages 53–64, 2008.

[10] Y.-W. Chen and C.-J. Lin. Combining svms with various feature selection strategies. In *Feature extraction*, pages 315–324. Springer, 2006.

[11] W. W. Cohen. Fast effective rule induction. In *ICML*, 1995.

[12] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12(Aug):2493–2537, 2011.

[13] C. Cortes and V. Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[14] T. M. Cover and J. A. Thomas. *Elements of information theory*. John Wiley & Sons, 2012.

[15] K. Crammer and Y. Singer. On the algorithmic implementation of multiclass kernel-based vector machines. *Journal of Machine Learning Research*, 2, 2001.

[16] V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 433–436. ACM, 2007.

[17] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American society for information science*, 41(6):391, 1990.

[18] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

[19] R. Duda, P. Hart, and D. Stork. *Pattern Classification*. Wiley Interscience, 2000.

[20] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 313–324. ACM, 2014.

[21] G. Gay, S. Haiduc, A. Marcus, and T. Menzies. On the use of relevance feedback in ir-based concept location. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 351–360. IEEE, 2009.

[22] A. Graves, A.-r. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*, pages 6645–6649. IEEE, 2013.

[23] A. Graves, G. Wayne, and I. Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[24] L. Hiew. *Assisted detection of duplicate bug reports*. PhD thesis, The University Of British Columbia, 2006.

[25] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[26] M. K. Hossen, H. Kagdi, and D. Poshyvanyk. Amalgamating source code authors, maintainers, and change proneness to triage change requests. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 130–141, 2014.

[27] X. Huo, M. Li, and Z. Zhou. Learning unified features from natural and programming languages for locating buggy source code. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 1606–1612, 2016.

[28] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, volume 1, pages 2073–2083.

[29] N. Jalbert and W. Weimer. Automated duplicate detection for bug tracking systems. In *DSN*, 2008.

[30] W. Jin and A. Orso. F3: fault localization for field failures. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 213–223. ACM, 2013.

[31] K. S. Jones. *Readings in information retrieval*. Morgan Kaufmann, 1997.

[32] N. Kalchbrenner, E. Grefenstette, and P. Blunsom. A convolutional neural network for modelling sentences. *arXiv preprint arXiv:1404.2188*, 2014.

[33] S. Kambhampati, editor. *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*. IJCAI/AAAI Press, 2016.

[34] F. Khomh, B. Chan, Y. Zou, and A. E. Hassan. An entropy evaluation approach for triaging field crashes: A case study of mozilla firefox. In *WCRE*, 2011.

[35] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering*, pages 489–498. IEEE Computer Society, 2007.

[36] Y. Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[37] P. S. Kochhar, Y. Tian, and D. Lo. Potential biases in bug localization: Do they matter? In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 803–814, 2014.

[38] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[39] K. Kurach, M. Andrychowicz, and I. Sutskever. Neural random-access machines. *arXiv preprint arXiv:1511.06392*, 2015.

[40] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports (n). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 476–481. IEEE, 2015.

[41] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals. Predicting the severity of a reported bug. In *MSR*, 2010.

[42] A. Lamkanfi, S. Demeyer, Q. Soetens, and T. Verdonck. Comparing mining algorithms for predicting the severity of a reported bug. In *CSMR*, 2011.

[43] J. L. Lawall, J. Brunel, N. Palix, R. R. Hansen, H. Stuart, and G. Muller. Wysiwib: A declarative approach to finding api protocols and bugs in linux code. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 43–52. IEEE, 2009.

[44] T. B. Le, R. J. Oentaryo, and D. Lo. Information retrieval and spectrum based bug localization: better together. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 579–590, 2015.

[45] T.-D. B. Le, S. Wang, and D. Lo. Multi-abstraction concern localization. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 364–367. IEEE, 2013.

[46] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[47] C.-P. Lee and C.-b. Lin. Large-scale linear ranksvm. *Neural computation*, 26(4):781–817, 2014.

[48] X. Li and B. Liu. Learning to classify texts using positive and unlabeled data. In *IJCAI*, volume 3, pages 587–592, 2003.

[49] X. V. Lin, C. Wang, D. Pang, K. Vu, L. Zettlemoyer, and M. D. Ernst. Program synthesis from natural language using recurrent neural networks. Technical Report UW-CSE-17-03-01, University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Mar. 2017.

[50] M. Linares-Vásquez, K. Hossen, H. Dang, H. Kagdi, M. Gethers, and D. Poshyvanyk. Triaging incoming change requests: Bug or commit history, or code authorship? In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, 2012.

[51] T.-Y. Liu. Learning to rank for information retrieval. *Foundations and Trends in Information Retrieval*, 3(3):225–331, 2009.

[52] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Source code retrieval for bug localization using latent dirichlet allocation. In *2008 15th Working Conference on Reverse Engineering*, pages 155–164. IEEE, 2008.

[53] D. MacKenzie, P. Eggert, and R. Stallman. *Comparing and Merging Files with GNU diff and patch.* Network Theory Ltd., 2003.

[54] C. D. Manning, P. Raghavan, and H. Schtze. *Introduction to Information Retrieval.* Cambridge University Press, 2008.

[55] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 214–223. IEEE, 2004.

[56] T. Menzies and A. Marcus. Automated severity assessment of software defect reports. In *ICSM*, 2008.

[57] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *Software Maintenance, 2000. Proceedings. International Conference on*, pages 120–130. IEEE, 2000.

[58] L. Moreno, J. J. Treadway, A. Marcus, and W. Shen. On the use of stack traces to improve text retrieval-based bug localization. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 151–160, 2014.

[59] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pages 1287–1293. AAAI Press, 2016.

[60] G. Murphy and D. Cubranic. Automatic bug triage using text categorization. In *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*. Citeseer, 2004.

[61] 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 modeling. In *ASE*, 2012.

[62] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Recurring bug fixes in object-oriented programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 315–324. ACM, 2010.

[63] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in linux device drivers. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 247–260. ACM, 2008.

[64] Y. Padioleau, J. L. Lawall, and G. Muller. Understanding collateral evolution in linux device drivers. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 59–71. ACM, 2006.

[65] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in linux: ten years later. In *ACM SIGPLAN Notices*, volume 46, pages 305–318. ACM, 2011.

[66] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang, and Z. Jin. Building program vector representations for deep learning. In *International Conference on Knowledge Science, Engineering and Management*, pages 547–553. Springer, 2015.

[67] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In *EMNLP*, volume 14, pages 1532–1543, 2014.

[68] 2011. www.ils.unc.edu/∼keyeg/java/porter/PorterStemmer.java.

[69] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33(6):420–432, 2007.

[70] M. Pradel and T. R. Gross. Leveraging test generation and specification mining for automated bug detection without false positives. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 288–298. IEEE, 2012.

[71] P. K. Prasetyo, D. Lo, P. Achananuparp, Y. Tian, and E.-P. Lim. Automatic classification of software related microblogs. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 596–599. IEEE, 2012.

[72] T. Qin, T.-Y. Liu, X.-D. Zhang, D.-S. Wang, W.-Y. Xiong, and H. Li. Learning to rank relational objects and its application to web search. In *Proceedings of the 17th international conference on World Wide Web*, pages 407–416, 2008.

[73] S. Rao and A. Kak. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 43–52. ACM, 2011.

[74] S. Robertson, H. Zaragoza, and M. Taylor. Simple bm25 extension to multiple weighted fields. In *Proceedings of the thirteenth ACM international conference on Information and knowledge management*, pages 42–49, 2004.

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

[76] H. Schütze. Introduction to information retrieval. In *Proceedings of the international communication of association for computing machinery conference*, 2008.

[77] K. Sen. Dart: Directed automated random testing. In *Haifa Verification Conference*, volume 6405, page 4, 2009.

[78] A. Sharma, Y. Tian, and D. Lo. Nirmal: Automatic identification of software relevant tweets leveraging language model. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 449–458. IEEE, 2015.

[79] A. Sharma, Y. Tian, and D. Lo. What's hot in software engineering twitter space? In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 541–545. IEEE, 2015.

[80] A. Sharma, Y. Tian, A. Sulistya, D. Lo, and A. F. Yamashita. Harnessing twitter to support serendipitous learning of developers. In *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*, pages 387–391. IEEE, 2017.

[81] Y. Shen, X. He, J. Gao, L. Deng, and G. Mesnil. Learning semantic representations using convolutional neural networks for web search. In *Proceedings of the 23rd International Conference on World Wide Web*, pages 373–374. ACM, 2014.

[82] R. Shokripour, J. Anvik, Z. M. Kasirun, and S. Zamani. Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 2–11, 2013.

[83] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *ACM sigsoft software engineering notes*, volume 30, pages 1–5. ACM, 2005.

[84] K. Small, B. Wallace, T. Trikalinos, and C. E. Brodley. The constrained weight space svm: learning with ranked features. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 865–872, 2011.

[85] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang. Towards more accurate retrieval of duplicate bug reports. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 253–262, 2011.

[86] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. 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*, pages 45–54, 2010.

[87] D. Surian, Y. Tian, D. Lo, H. Cheng, and E.-P. Lim. Predicting project outcome leveraging socio-technical network patterns. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, pages 47–56. IEEE, 2013.

[88] 2011. http://svmlight.joachims.org/svm_multiclass.html.

[89] A. Tamrawi, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen. Fuzzy set and cache-based approach for bug triaging. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 365–375, 2011.

[90] Y. Tian, P. Achananuparp, I. N. Lubis, D. Lo, and E.-P. Lim. What does software engineering community microblog about? In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, pages 247–250. IEEE, 2012.

[91] Y. Tian, N. Ali, D. Lo, and A. E. Hassan. On the unreliability of bug severity data. *Empirical Software Engineering*, 21(6):2298–2323, 2016.

[92] Y. Tian, P. S. Kochhar, and D. Lo. An exploratory study of functionality and learning resources of web apis on programmableweb. In *International Conference Evaluation and Assessment in Software Engineering*, 2017.

[93] Y. Tian and D. Lo. Leveraging web 2.0 for software evolution. In *Evolving Software Systems*, pages 163–197. Springer, 2014.

[94] Y. Tian and D. Lo. A comparative study on the effectiveness of part-of-speech tagging techniques on bug reports. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 570–574. IEEE, 2015.

[95] Y. Tian, D. Lo, and J. Lawall. Automated construction of a software-specific word similarity database. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 44–53. IEEE, 2014.

[96] Y. Tian, D. Lo, and J. Lawall. Sewordsim: Software-specific word similarity database. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 568–571. ACM, 2014.

[97] Y. Tian, D. Lo, and C. Sun. Information retrieval based nearest neighbor classification for fine-grained bug severity prediction. In *WCRE*, 2012.

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

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

[100] J.-M. Torres-Moreno. *Automatic text summarization*. John Wiley & Sons, 2014.

[101] S. Wang, T. Liu, and L. Tan. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering*, pages 297–308. ACM, 2016.

[102] S. Wang, D. Lo, and J. Lawall. Compositional vector space models for improved bug localization. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 171–180, 2014.

[103] S. Wang, D. Lo, Z. Xing, and L. Jiang. Concern localization using information retrieval: An empirical study on linux kernel. In *2011 18th Working Conference on Reverse Engineering*, pages 92–96. IEEE, 2011.

[104] X. Wang, D. Lo, X. Xia, X. Wang, P. S. Kochhar, Y. Tian, X. Yang, S. Li, J. Sun, and B. Zhou. Boat: an experimental platform for researchers to comparatively and reproducibly evaluate bug localization techniques. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 572–575. ACM, 2014.

[105] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of the 30th international conference on Software engineering*, pages 461–470, 2008.

[106] http://www.cs.waikato.ac.nz/ml/weka/. Weka 3: Data Mining Software, 2011.

[107] C. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 181–190, 2014.

[108] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung. Relink: recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 15–25, 2011.

[109] X. Xia, D. Lo, S. J. Pan, N. Nagappan, and X. Wang. Hydra: Massively compositional model for cross-project defect prediction. *IEEE Transactions on Software Engineering*, 42(10):977–998, 2016.

[110] X. Xuan, D. Lo, X. Xia, and Y. Tian. Evaluating defect prediction approaches using a massive set of metrics: An empirical study. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1644–1647. ACM, 2015.

[111] X. Ye, R. C. Bunescu, and C. Liu. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 689–699, 2014.

[112] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 14–24. IEEE, 2012.