Analysis of Human Affect and Bug Patterns to Improve Software Quality and Security

Md Rakibul Islam
mislam3@uno.edu

Analysis of Human Affect and Bug Patterns to Improve Software Quality and Security

A Dissertation

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Engineering and Applied Science
Computer Science

by

Md Rakibul Islam

B.S. Khulna University, 2008

May, 2020

This dissertation is dedicated to my parents
MST. BILKIS BEGUM and
late ABUL KALAM AZAD

Acknowledgements

First and foremost, I cordially express my gratitude and heartfelt thanks to my adviser Dr. Minhaz F. Zibran for allowing me to pursue my PhD under his supervision. Dr. Zibran has always been very supportive, caring, inspirational, and motivational. His innovative ideas and proper guiding immensely helped me to keep myself in the right direction of my PhD journey.

I also thank the members of my thesis advisory committe, Dr. Md Tamjidul Hoque, Dr. Shengru Tu, Dr. Linxiong Li, and Dr. Syed Adeel Ahmed who have been very supportive by providing their invaluable assistance, and feedback at all stages of this thesis. Their criticisms, comments, and advice are critical in making this dissertation more accurate and complete. Thanks to the anonymous reviewers for their valuable comments and suggestions in improving the work and publications produced from this thesis.

I am thankful to my fellow researchers, faculty members, and staff of the Department of Computer Science for their spontaneous cooperation and encouragement. I especially thank Aayush Nagpal, Md Kauser Ahmmed, Pradeep Jakibanjar, and Rahul Chatterjee with whom I jointly worked on a couple of research projects. I would also like to thank the University of New Orleans for providing me an excellent environment for research and the financial support.

My father who dreamed and believed that I could come this far to complete my PhD. Unfortunately, he is no more with us to see me fulfilling his dream. I will never be able to say to him "I have done it because of you, and I miss you a lot". My mother always remains by my side to give me love, courage, guidance, and everything I need to overcome any difficult situations whenever that come on my ways. I am highly obliged to my parents, and acknowledge their significant contributions in my every achievement.

I am also grateful to my wife MST. Shahana Islam my soul-mate and mother of our son Surahbil Sabib Tasfi. Living in a foreign country would never had been so easy without a caring partner like her. She has been extremely supportive of me throughout my entire PhD journey and has made countless sacrifices to help me get to this point. She always has tried to ease my stresses with her sweet smile and love. I believe, when our son *Tasfi* will be a grown-up, he will be proud by realizing what his father has achieved, and how he has been a source of inspiration in achieving that.

Last but not least, I extend my special thanks to my sister, uncles, other family members, and my friends Rajeeb, Reja, Jubayer, Monir, Shafiqur, Atik, Pulok, Ratul, Rana, and others at UNO who helped me in many ways, and gave me many moments of joy.

Table of Contents

List of Figures

List of Tables

Abstract

The impact of software is ever increasing as more and more systems are being software operated. Despite the usefulness of software, many instances software failures have been causing tremendous losses in lives and dollars. Software failures take place because of bugs (i.e., faults) in the software systems. These bugs cause the program to malfunction or crash and expose security vulnerabilities exploitable by malicious hackers.

Studies confirm that software defects and vulnerabilities appear in source code largely due to the human mistakes and errors of the developers. Human performance is impacted by the underlying development process and human affects, such as *sentiment* and *emotion*. This thesis examines these human affects of software developers, which have drawn recent interests in the community. For capturing developers' sentimental and emotional states, we have developed several software tools (i.e., `SentiStrength-SE`, `DEVA`, and `MarValous`). These are novel tools facilitating automatic detection of sentiments and emotions from the software engineering textual artifacts.

Using such an automated tool, the developers' sentimental variations are studied with respect to the underlying development tasks (e.g., bug-fixing, bug-introducing), development periods (i.e., days and times), team sizes and project sizes. We expose opportunities for exploiting developers' sentiments for higher productivity and improved software quality.

While developers' sentiments and emotions can be leveraged for proactive and active safeguard in identifying and minimizing software bugs, this dissertation also includes in-depth studies of the relationship among various *bug patterns*, such as *software defects*, *security vulnerabilities*, and *code smells* to find actionable insights in minimizing software bugs and improving software quality and security. Bug patterns are exposed through mining software repositories and bug databases. These bug patterns are crucial in localizing bugs and security vulnerabilities in software codebase for fixing them, predicting portions of software susceptible to failure or exploitation by hackers, devising techniques for automated program repair, and avoiding code constructs and coding idioms that are bug-prone.

The software tools produced from this thesis are empirically evaluated using standard measurement metrics (e.g., precision, recall). The findings of all the studies are validated with appropriate tests for statistical significance. Finally, based on our experience and in-depth analysis of the present state of the art, we expose avenues for further research and development towards a holistic approach for developing improved and secure software systems.

Keywords: Software Engineering; Human affects; Sentiment; Emotion; Software Bug; Software Security; Code Quality.

Chapter 1

Introduction and Motivation

Software systems have become ubiquitous these days, and technology rarely exists today without a software interface or component. The impact of software is ever increasing as more and more systems are being software operated. Since the emergence of software systems, many instances of software failures have been causing tremendous losses in lives and dollars. Software failures take place because of bugs (i.e., faults and security issues) in the software systems. The battle against software bugs exists since software existed.

One of the main reasons why bugs exist is disappointingly simple: software is developed by humans and humans are prone to error. Studies [1, 2] also confirm human error as one of the primary causes of software defects. As a primary cause of software defects, human error can be the key in understanding and minimizing software defects [2]. Surprisingly, much of software engineering research in the last decade is technical and deemphasizes the human factors [3, 4, 2] that are mainly liable for human error. In this thesis, we mine and analyze one important human factor, i.e., human affect (e.g., feelings, sentiments, and emotions) that can be exploited in minimizing and identifying human error to develop high-quality software systems.

Although companies, such as Facebook, Google, and Microsoft take various initiatives to ensure developers' positive affective states to minimize human errors, it can not make sure error/mistake free programming. Thus, companies also take many measurements at source code level, such as software testing [5, 6, 7, 8], and source code analysis (e.g., bug localization) [9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21] to tackle software bugs. Despite taking those measurements, a great amount of software systems are regularly shipped with both known and unknown defects [22]. Indeed, there are more bugs in real-world programs than human programmers can realistically address [23]. Thus, in this thesis, we also analyze source code to identify actionable insights to minimize software bugs.

According to psychology, interestingly, far from being random, humans' mistakes tend to fall into recurrent patterns [24]. Developers, being human, frequently exhibit patterns in making decisions and solving problems, and also in the errors/mistakes committed by them. Mistakes of similar patterns are fixed in similar fashions, i.e., patterns exist in the mistakes' fixings too. Knowing and analyzing the patterns of the mistakes and their fixings can help developers in avoiding those mistakes, and also in generating solutions automatically for them. Thus, we also mine and analyze patterns in fixings of *developers' mistakes that result in software bugs*.

While taking cares of developers' affects, and making them aware of the patterns of the common mistakes can minimize software bugs, and improve software quality, there will be still unexpected issues in source code. Developers' malpractices during development can lead to software bugs and poor code quality. Developers often apply *copy-paste* technique when they are stressed (due to time pressure), or inexperienced. However, the *copy-paste* technique is dangerously a bad practice [25] since that may propagate software bugs, security vulnerabilities and other issues that hamper code quality. Thus, we also mine and analyze developers' copy-paste practice to identify its impacts on software bugs and code quality.

In this thesis, *we take on a holistic approach to mine and analyze (i) human affects expressed in natural language texts (e.g., commit comments), and (ii) patterns of bugs identified in source code to gain actionable insights that can be leveraged in minimizing bugs and improving quality and security of software systems*. In Figure 1.1, we pictorially summarize the research problem and divide that into sub-problems that we address in this thesis. In the following, we also elaborately discuss the research sub-problems, and briefly mention our contributions with respect to the sub-problems.

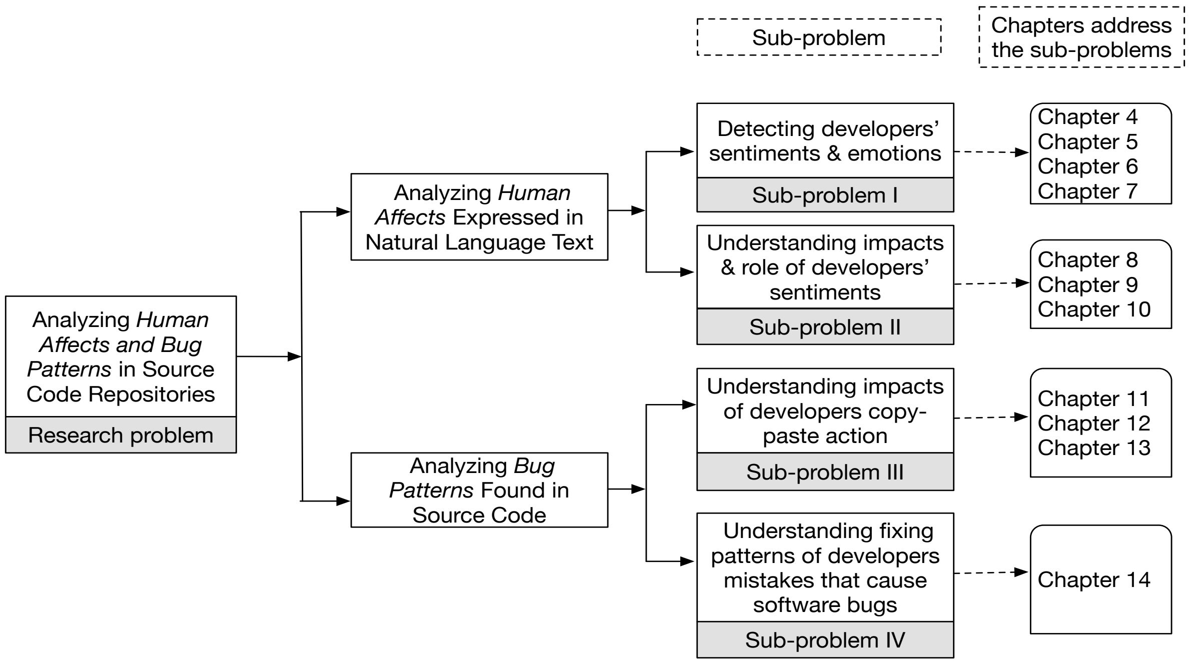

Sub-problem
Chapters address the sub-problems
Analyzing Human Affects and Bug Patterns in Source Code Repositories
Research problem
Analyzing Human Affects Expressed in Natural Language Text
Analyzing Bug Patterns Found in Source Code
Detecting developers' sentiments & emotions
Sub-problem I
Understanding impacts & role of developers' sentiments
Sub-problem II
Understanding impacts of developers copy-paste action
Sub-problem III
Understanding fixing patterns of developers mistakes that cause software bugs
Sub-problem IV
Chapter 4
Chapter 5
Chapter 6
Chapter 7
Chapter 8
Chapter 9
Chapter 10
Chapter 11
Chapter 12
Chapter 13
Chapter 14

Figure 1.1: Research problem and thesis organization

1.1 Mining and Analyzing Human Affect in Software Engineering

Sentiments influence people's activities and interactions, and thus sentiments affect task quality, productivity, creativity, group rapport, and job satisfaction [26, 27, 28]. Software development, being highly dependent on human efforts and interactions, is more susceptible to sentiments of the practitioners. Hence, a good understanding of the developers' sentiments and their influencing factors can be exploited for effective collaborations, task assignments [29], and in devising measures to boost up job satisfaction, which, in turn, can result in increased code quality and productivity [30], and reduced software bugs.

Attempts are made to capture the developers' sentiments in the workplace by means of traditional approaches such as interviews, surveys [31], and biometric measurements [32]. Capturing sentiments with the traditional approaches is more challenging for projects relying on geographically distributed team settings and voluntary contributions (e.g., open-source projects) [33, 34]. Moreover, the traditional approaches involving direct observations and interactions with the developers often hinder their natural workflow. Thus, to supplement or complement those traditional approaches, recent attempts detect sentiments from the software engineering textual artifacts such as issue comments [35, 36, 34, 37, 38, 39, 40, 41], email contents [42, 43], and forum posts [4, 44].

1.1.1 Sub-problem I: Detecting developers' sentiments and emotions and emotions with high accuracy

For automated extraction of sentiments from textual artifacts in the software engineering domain, three tools (i.e., `SentiStrength` [45], `NLTK` [46], and `Stanford NLP` [47]) are used while the use of `SentiStrength` is found dominant [48, 49]. However, software engineering studies [35, 36, 37, 48, 50, 41, 51, 43] involving sentiment analysis repeatedly *report concerns about the accuracy of those sentiment analysis tools in the detection of sentimental polarities (i.e., negativity, positivity, and neutrality) of plain text contents*. For example, when applied in the software engineering domain, `SentiStrength` and `NLTK` are respectively reported to have only 29.56% and 52.17% precision in identifying positive sentiments, and even lower precision of 13.18% and 23.45% respectively in the detection of negative sentiments [48, 43].

Those sentiment analysis tools are developed and trained using data from non-technical social networking media (e.g., twitter posts, forum posts, movie reviews) and when operated in a technical domain such as software engineering, their accuracy substantially degrades largely. *Thus, the software engineering community demands a more accurate automatic sentiment analysis tool* [35, 36, 38, 48, 50, 52, 41, 53, 43].

1.1.2 Contribution

To address the sub-problem I, first, using a large benchmark dataset, we carry out an in-depth exploratory study for exposing the difficulties in automatic sentiment analysis in textual content in a technical domain such as software engineering. To the best of our knowledge, this is the first investigation conducted on a public benchmark dataset to identify challenges of sentiment analysis in software engineering. Then, we propose techniques and realize those in `SentiStrength-SE`, a prototype tool that we develope for improved sentiment analysis in software engineering textual content. The tool is also made freely available online [54]. We further conduct a qualitative evaluation of our tool. Based on the exploratory study and the qualitative evaluation, we outline plans for further improvements in automated sentiment analysis in the software engineering area.

We also compare the performance of `SentiStrenghth-SE` against its latter developed domain-specific counterparts (e.g., `Senti4SD` [55], `EmoTxt` [56]) to identify the strengths and weaknesses of the domain-specific tools for future considerations. Finally, we select four dictionaries, developed using four distinct methods, to quantitatively compare their sentiment detection performances in software engineering textual artifacts [57] to identify which methods have higher/lower potential to perform well in constructing a domain dictionary to fit in `SentiStrenghth-SE`.

Then, we develop another tool `DEVA` [58] that is capable of detecting four emotional/sentimental states such as *excitement, stress, depression,* and *relaxation*. For empirical evaluation of `DEVA`, we construct a ground-truth dataset consisting of 1,795 *JIRA* issue comments, each of which are manually annotated by three human raters. This tool along with the ground-truth dataset are also made freely available online [59].

Next, we develop `MarValous`, which is the first Machine Learning (ML) based tool that performs significantly better than `DEVA` in detecting individual emotional states *excitation, stress, depression, relaxation* and *neutrality* from software engineering texts. By using nine preprocessing steps and seven features, we have developed the tool `MarValous` that consists of nine popular and effective supervised ML algorithms for emotion/sentiment analysis.

1.1.3 Sub-problem II: Understanding developers' sentiments

Few studies have been performed in the past for understanding the role of sentiments on software development and engineering. Some of those earlier studies address *when* and *why* employees get affected by sentiments [26, 34, 4, 41, 43], whereas some other work address *how* [60, 61, 62, 63, 31] the sentiments impact the employees' performance at work.

Despite few earlier studies, software engineering practices still lack theories and methodologies for addressing human factors including sentiments [3, 4]. *Hence, the community calls for research on the roles and impacts of sentiments in software engineering* [61, 28, 64].

1.1.4 Contribution

In this thesis, we conduct a quantitative empirical study of the sentimental variations in different types of development activities (e.g., bug-fixing tasks), development periods (i.e., days and times) and in projects of different sizes involving teams of variant sizes [37, 38]. The study also includes an in-depth investigation of sentiments' impacts on software artifacts (i.e., commit messages) and exploration of scopes for exploiting sentimental variations in software engineering activities. The findings from this work add to our understanding of the role of sentiments in software development and expose scopes for exploitation of sentimental awareness in improved task assignments and collaborations, which ultimately result in high-quality software.

We also study the sentimental variations in bug-introducing and bug-fixing commit messages [65]. The results advance our understanding of the extent to which sentiments affect tasks that result software bugs, and how such bug-introducing commits differ from bug-fixing ones in terms of the sentiments expressed in the commit messages.

1.2 Mining and Analyzing Impacts of Developers' Copy-Paste Actions

Developers often reuse existing code by copy-paste to increase their productivity. Such a reuse mechanism typically results in duplicate or very similar code fragments commonly known as *code clones*. Aside from such deliberate cloning, unintentional clones are also created for various reasons under diverse circumstances [66, 67]. Software systems typically have 9%-17% [68] cloned code, and the proportion is sometimes found to be even 50% [69] or higher [70].

Despite few benefits [71] of cloning, code clones are detrimental in most cases [71, 72, 73]. Code clone is a notorious code smell (i.e., a symptom indicating source of future problems) [74], that cause serious problems such as *reduced code quality*, *code inflation*, *program faults*, *security vulnerabilities*, and *bugs propagation* [72, 75]. Clones are thus a major contributor to the high maintenance cost for software systems, and as much as 80% of software costs are spent on maintenance [76]. Therefore, it is necessary to keep the number of clones at the minimum and to remove them from source code by refactoring. However, not all the clones in a software system are harmful [71], neither it is feasible to remove all the clones in source code by refactoring [77, 75]. Therefore, we must distinguish the context and characteristics of clones, which make them malign as opposed to the benign clones.

Towards this goal, several studies have been performed in the past to examine or exploit comparative stability of clones as opposed to non-cloned code [78, 79, 80, 81, 82], relationships of clones with bug-fixing changes [25, 72, 83, 84, 85, 86, 87, 88, 89], and the impacts of clones on program's changeability [73, 90, 91].

1.2.1 Sub-problem III: Understanding impacts of developers copy-paste action

Earlier attempts [85, 86, 87, 89] to determine detrimental impacts of developers copy-paste actions, that result in code clones, relied on long-term history of bug fixing changes preserved in version control system. While such studies make important contributions, *their approaches do not fit well for proactive clone management, especially at the early stages of software development process where significantly long history of bug-fixing changes are not available. Therefore, to determine the* *detrimental impacts of code clones, we need to apply static source code analysis techniques that do not require any bug-fixing history.*

Moreover, software security has become one of the most pressing concerns recently. *However, the security aspects of code clones have never been studied before, although copy-paste of vulnerable components and source code (i.e., code clones) multiply security vulnerabilities* [92, 93, 94].

1.2.2 Contribution

To gain insights into detrimental implications of code cloning, we use static code analysis techniques to mine *code smells* and *security vulnerabilities* and study them in cloned and non-clone code [95, 96]. Using statistical analyses, we derive insights into how the code smells and security vulnerabilities cause software bugs and impact code quality of cloned code compared to non-clone code. We also conduct another study on the characteristics of buggy clones from a code quality perspective [97].

Findings from the earlier studies add to our understanding of the characteristics and impacts of clones, which can be useful in clone-aware software development and in devising techniques for minimizing the negative impacts (e.g., software bugs) of code clones and increasing code quality.

1.3 Mining and Analyzing Fixing Patterns of Developers' Mistakes

We also mine and analyze fixing patterns of developers' mistakes that cause software bugs. Efforts to fix bugs consume a vast amount of total expenses in software maintenance [98] while nearly 80% of software cost is spent in maintenance [76]. Typical bug-fixing efforts mainly involve two types of tasks: (a) localization of bugs in source code, and (b) bug-fixing edits to the source code. A deep understanding of the common bug-fixing patterns can immensely help in minimizing efforts in both of these tasks and also contribute to devising techniques for automated program repair (APR). A good understanding of the bug-fixing patterns can also help a developer to proactively avoid writing code that leads to program faults.

1.3.1 Sub-problem IV: Understanding fixing patterns of software bugs

Earlier studies on discovering bug-fix patterns remained focused on bug-fixing *edit* patterns, which include bug-fixing changes to the source code at a very fine-grained level without capturing the nesting levels of the surrounding code. *Thus, earlier studies are limited in two ways: (i) a very fine-grained level presentation of patterns makes it difficult to perceive easily by a human, and (ii) those studies missed nested code structure (i.e.,* nesting *patterns), which is an important aspect of bug-fix patterns.*

1.3.2 Contribution

In this work, we capture both bug-fixing edit patterns and nesting patterns (i.e., frequent nested code structures) of bug-fixing edits through an in-depth (quantitative and qualitative) analysis of 4,653 buggy revisions of five software systems drawn from diverse application domains. We identify a total of 38 bug-fix patterns organized in 14 categories. This is the highest number of bug-fix patterns identified in a single study. Four of these patterns are completely new, and 34 of them confirm those reported in earlier studies. We also study locations of bug-fix changes in nested code structures and identify 37 nesting patterns that hold the majority of the bug-fix edits. These nesting patterns are new (i.e., never targeted before), and add a new dimension in our understanding of bug-fix patterns.

1.4 Outline of the Thesis

In this chapter (Chapter 1), we have introduced the research problem and its sub-problems along with their backgrounds. We also briefly mentioned the contributions with respect to the sub-problems. The remaining of this thesis is organized as follows. Chapter 2 presents the terminologies and concepts that develop the background to follow the remaining of the thesis. Chapter 3 describes the research methodology where we briefly describe each work with respect to the problem and motivation it addresses, and how we solve the problem.

Chapter 4 to Chapter 14 are related to the respective sub-problems as shown in Figure 1.1. In Chapter 4, we identify the challenges of detecting sentiment from software engineering texts and describe how the identified challenges are addressed to develop the first software engineering domain specific sentiment analysis tool `SentiStrength-SE`. We compare the performance of `SentiStrength-SE` tool against its counterparts in Chapter 5. In Chapter 6, we present the first emotions analysis tool, `DEVA`, which is especially designed for software engineering text to capture *excitation*, *stress*, *depress*, and *relaxation*. Later, using machine learning and natural language processing techniques, we develop the tool `MarValous`, which is an improved version of `DEVA`. The development procedure of `MarValous` and its evaluation results are presented in Chapter 7.

Chapter 8 presents a quantitative empirical study of the sentiment variations in different types of development activities (e.g., bug-fixing tasks) and development periods (i.e., days and times), in addition to in-depth investigation of emotions' impacts on software artifacts (i.e., commit messages) and exploration of scopes for exploiting emotional variations in software engineering activities. In Chapter 9, we present a study of sentiment variations in bug-introducing and bug-fixing commit messages that helps in understanding of the extent to which sentiments impact software development tasks. In Chapter 10, we identify the roles of human affects, i.e., sentiments and emotions in various software engineering activities.

In Chapter 11, we describe a comparative study on different types of clones (i.e., copy-pasted code) and non-cloned code on the basis of their code-smells. To explore and understand the security vulnerabilities and their severity in different types of clones compared to non-clone code, we present a study in Chapter 12. Chapter 13 presents a comparative study to distinguish the characteristics (from a code quality perspective) of buggy and non-buggy clones. Chapter 14 presents an in-depth quantitative and qualitative analysis of buggy revisions in understanding of the common patterns of bug-fixing changes. Finally, Chapter 15 concludes this thesis.

Major parts of this thesis are already published in peer reviewed journals and international conferences. A list of publications that are outcomes of this dissertation research are presented in Appendix A.

Chapter 2

Background

In this chapter, we introduce the terminologies and concepts necessary to develop the background that will help to follow the remaining of this thesis. We begin with the definitions of software engineering terminologies in Section 2.1. Section 2.2 and Section 2.3 respectively describe the terminologies and concepts related to sentiments and developers' copy-paste practice. In Section 2.4 and Section 2.5, we respectively introduce *code smells* and *security vulnerabilities* that can be spread in source code due to developers' copy-paste practice. Finally, Section 2.6 summarizes the chapter.

2.1 Software Engineering Terminologies

Software Bug: A software bug is an error, flaw, failure or fault in a computer program or system that causes it to produce an incorrect or unexpected result, or to behave in unintended ways.

Issue Repository: An issue repository provides a great facility to describe and keep track of any type of issue such as tasks, enhancements, and bugs for a software project. In an issue repository, an issue can be shared and discussed among the team members in written form.

Issue Comments: In an issue repository, all the descriptions of issues and discussions among team members (about those issues) happen in written forms, which are known as issue comments.

Version Control System: A version control system keeps track of every modification to the code in a special type of software repository. It helps a software team to manage changes to source code over time.

Code Commit: In version control systems, a *code commit* adds the latest code changes to a shared code repository.

Commit messages: While committing code in a version control system, a developer usually describes or illustrates his code using a textual message, which is known as a commit message or comment.

Bug-introducing commit: A type of code commit, which introduce a bug in software, is known as a bug-introducing commit.

Bug-fixing commit: A code commit that fixes a bug is known as a bug-fixing commit.

2.2 Sentiment Related Terminologies

Sentiment: According to the Merriam-Webster online dictionary [99], sentiment refers to an attitude, thought, or judgment prompted by feeling. In other words, sentiment is the attractiveness (or adverseness) to an event, object, or situation [100].

There are mainly two types of sentiments as follows.

1. *Positive* sentiment, which expresses positive feeling toward an object or action. As for example, in the following text, the writer expresses positive sentiment.

   Example: I love to fix the bug in the software.

2. *Negative* sentiment, which expresses negative feeling toward an object or action. As for example, in the following text, the writer expresses negative sentiment.

   Example: The bug seems very nasty to solve.

A feeling without sentiment is *neutral* in sentiment.

Emotion: Emotions are the expressions of affect and/or feelings. Similar to this thesis, in many studies, the concepts of *emotions*, *sentiments*, *feeling* and *affect* are also used interchangeably [101, 102].

Arousal: Arousal represents the intensity of emotional activation [103]. It is the sensation of being mentally awake and reactive to stimuli, i.e., vigor and energy or fatigue and tiredness [104]. In the following example, the three exclamation signs at the end of the comment, express high *arousal* of the comment writer.

Example: I am very happy to see it is useful and used !!!."

2.2.1 Dimensional Framework of Emotions

Here we present a simple bi-dimensional model [105, 61] of emotions, which is a variant of the dimensional framework, commonly known as VAD (aka PAD) model [106]. In the bi-dimensional model, as shown in Figure 7.1, the horizontal dimension presents the sentimental *polarities* also known as *valence* and the vertical dimension indicates the levels of *arousal*.

The dimensions are bipolar where the valence dimension ranges from negative to positive and the arousal dimension ranges from low to high. Many emotional states of a person can be determined by combining valence and arousal. For example, positive valence and high arousal, in combination, indicate the emotional/sentimental state *excitement*. However, instead of deriving many emotional states, for simplicity a large number studies use a set of four major classes of emotional states that include *excitement, stress, depression,* and *relaxation* [61, 107].

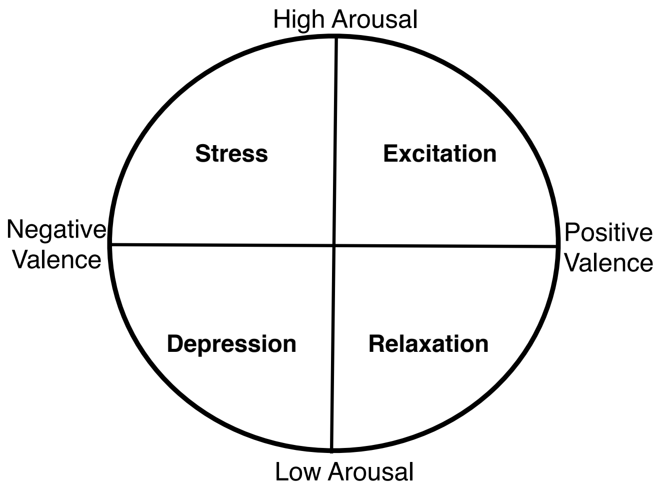


Figure 2.1: Simple bi-dimensional model of emotions

As shown in Figure 7.1, the four emotional states are very distinct, as each state constitutes emotions, which are quite different compared to the emotions of other states [105]. Thus, the model is unequivocal to recognize emotions, simple and easy to understand.

2.2.2 A Popular Sentiment Analysis Technique in Software Engineering

Sentiment analysis using a lexical based technique on a given piece of text (e.g., a commit message) c computes a pair $\langle \rho_c, \eta_c \rangle$ of integers, where $+1 \leq \rho_c \leq +5$ and $-5 \leq \eta_c \leq -1$. Here, ρ_c and η_c respectively represent the positive and negative sentimental scores for the given text c.

A given text c is considered to have positive emotions if $\rho_c > +1$. Similarly, a text is held containing negative emotions when $\eta_c < -1$. Note that, a given text can exhibit both positive and negative emotions at the same time, and a text is considered emotionally neutral when the emotional scores for the text appear to be $\langle 1, -1 \rangle$.

Procedure to compute sentimental scores: A lexicon-based sentiment classifier (LSC) maintains a dictionary of several lists of words and phrases as its key dictionaries to compute sentiments in texts. Among these lists, the *sentimental words list, list of booster words, list of phrases*, and *list of negations words* play a vital role in the computation of sentiments. The entries in all these lists except the list of negation words are pre-assigned with sentimental scores. The negation words in the fourth list are used to invert the sentimental polarity of a term when the term is located after a negation word in a text.

For an input sentence, an LSC extracts individual words from the sentence and searches for each of the individual words in the *sentimental words list* to retrieve the corresponding sentimental scores. A similar search is made in the *list of booster words* to strengthen or weaken the sentimental scores.

Table 2.1: Role of a dictionary list in computation of sentimental scores in text

Sample Sentence	Sent. Score		Dictionary Lists in Use	Explanation
	ρ_c	η_c		
It's a *good* feature.	2	-1	Sentimental Words	The sentimental score of the word *'good'* is pre-assigned to 02; so the sentence is assigned positive score 02.
It's a *very good* feature.	3	-1	Booster Words	As booster word *'very'* is used before the sentimental word, the sentence is assigned a positive score 03
It's *not good* feature.	1	-2	Negations	Sentimental polarity of the sentimental word is inverted in here due to the use of the negation word *'not'* before sentimental word
It's a *killer feature*	2	-1	Phrases	*"killer feature"* is a phrase in the dictionary with positive score 02. Although the word *'kill'* carries negative sentiment, its effect is overridden by the sentimental score of the enclosing phrase

The *list of phrases* is used to distinguish groups of words as commonly used phrases. When such a phrase is identified, the sentimental score of the phrase overrides sentimental scores of the individual words, which constitute the phrase. The examples in Table 4.2 articulate how an LSC depends on a dictionary list for computing sentimental scores in plain texts.

2.3 Terminologies Related to Developers' Copy-paste Actions

As stated earlier, developers' copy-paste actions can result in duplicate or similar code fragments, which are roughly known to be code clones. In the following, we present the definitions of different types of code clone.

Type-1 Clones: Identical pieces of source code with or without variations in whitespaces (i.e., layout) and comments are called *Type-1* clones [67].

Type-2 Clones: *Type-2* clones are syntactically identical code fragments with variations in the names of identifiers, literals, types, layout and comments [67].

Type-3 Clones: Code fragments, which exhibit similarities as of *Type-2* clones and also allow further differences such as additions, deletions or modifications of statements are known as *Type-3* clones [67].

Notice that by the definitions above, *Type-2* clones include *Type-1* while *Type-3* clones include both *Type-1* and *Type-2*. Let, T_1, T_2, and T_3 respectively denote the sets of *Type-1*, *Type-2*, and *Type-3* clones in a software system. Mathematically, $T_1 \subseteq T_2 \subseteq T_3$. Thus, we further define two subsets of *Type-2* and *Type-3* clones as follows.

Pure Type-2 Clones: A set of pure *Type-2* clones include only those *Type-2* clones that do not exhibit *Type-1* similarity. Mathematically, $T_2^p = T_2 - T_1$, where T_2^p denotes the set of pure *Type-2* clones.

Pure Type-3 Clones: A set of pure *Type-3* clones include only those *Type-3* clones, which do not exhibit similarities at the levels of *Type-1* or *Type-2* clones. Mathematically, $T_3^p = T_3 - T_2$, where T_3^p denotes the set of pure *Type-3* clones.

2.3.1 Clone Granularity

The definitions of all three types of clones are based on the notion of the code segment, and contiguous portion of code at different levels of granularity have been used in the literature. As concerned with source code, the most commonly used granularities are at the level of the entire source file, class definition, method body, code block, and statements, which yield the notion of code clones of the following five types:

File clone: When two files are found to have contained similar enough source code, they are called file clones.

Class clone: Two classes of object-oriented source code can be considered as class clones if they have an identical or near-identical code.

Function clone: Two functions are considered as clones when the bodies of the functions consist of code that is similar enough.

Block clone: When two blocks of code (marked with opening and closing braces or indentation, or the like) are similar enough, they are called block clones. To consider clones at the granularity of blocks, one must decide how to deal with nested blocks (i.e., blocks inside another block).

Arbitrary statements clone: When two groups of statements at arbitrary regions of the source file are found to be similar enough, they are also regarded as clones.

2.4 Code Smell

Code smells are certain structures or patterns in code that indicate violations of fundamental design principles, which negatively impact design quality. Code smells are usually not bugs; they are not technically incorrect and do not prevent the program from functioning. Instead, they indicate weaknesses in design that may slow down development or increase the risk of bugs or failures in the future. Few commonly found code smells are listed and briefly describe in Table 11.1.

Table 2.2: Abridged Definitions of Common Code Smells

Code Smell	**Definition**
LawOfDemeter	Program unit needing too much knowledge about other units.
LocalVariableCouldBeFinal	Local variable assigned only once but not declared final.
ShortVariable	A field, local, or parameter with a too short name.
OnlyOneReturn	Method with more than one exit points.
IfStmtsMustUseBraces	'if' statements without accompanying curly braces.
AssertionsShouldIncludeMessage	Assertions including *no* error message.
UselessParentheses	Useless parentheses in code.
IfElseStmtsMustUseBraces	'if-else' statements without accompanying curly braces.
AvoidInstantiatingObjectsInLoops	Instantiation of new objects inside loop.

2.5 Security Vulnerabilities

A software security vulnerability is defined as a weakness in a software system that can lead to a compromise in integrity, availability or confidentiality of that software system. For example, *buffer overflow* and *dangling pointers* are two well-known security vulnerabilities. The cybersecurity community maintains a community-developed list of common software security vulnerabilities where each category of vulnerability is enumerated with a CWE (Common Weakness Enumeration) number [108]. For example, CWE-120 refers to those vulnerabilities that fall into the CWE category of *classic buffer overflow*. More examples of security vulnerabilities along with their CWE enumerations are presented in Table 12.1.

Table 2.3: Abridged Definitions of Common Security Vulnerabilities

Security Vulnerability	**Description**
Buffer Overflow	An application attempts to write data past the end of a buffer (CWE-120).
Uncontrolled Format String	Submitted data of an input string is evaluated as a command by the application (CWE-134).
Integer Overflow	The result of an arithmetic operation exceeds the maximum size of the integer type used to store it (CWE-190).
Null Pointer Dereference	Dereference a pointer that is null (CWE-476).
Memory Leak	Not release allocated memory (CWE-401).
Null Termination Errors	A string is incorrectly terminated (CWE-170).

2.6 Summary

In this chapter, we have introduced the terminologies and preliminary concepts that help in forming the foundation to understand the thesis. At the beginning of the chapter, we have presented different terminologies of software engineering. Then, we have introduced the terminologies related sentiments. We have also introduced the two-dimensional emotional model and discussed a popular lexical-based sentiment analysis technique. Latter, we have presented definitions of different types of code clone and granularity. We have also described how code clones can appear in the source code due to the developers' intention or even without their awareness. Finally, we have introduced code smell and security vulnerability along with examples.

Chapter 3

Research Methodology

In this chapter, we briefly describe the research methodologies followed to address the problem defined in Chapter I. Our research methodology for addressing the problem comprised of the following aspects: (i) describing a problem's background, (ii) our objective/solution, (iii) methodology to fulfill the objective/solution, and (iv) result.

To address sub-problem I (i.e., accurately detecting sentiment and emotion), we develop three state-of-the-art software engineering domain specific tools (i.e., `SentiSrength-SE`, `DEVA`, and `MarValous`), which are described in subsection 3.1.1, subsection 6.3, and subsection 3.1.5 respectively. We further address sub-problem I by comparing our `SentiSrength-SE` against its domain specific counterparts in subsection 3.1.2. In subsection 3.1.3, we compare four dictionaries, developed using four distinct methods, to compare their sentiment detection performances in software engineering textual artifacts.

To address sub-problem II, respectively in subsection 3.1.6 and subsection 3.1.7, we describe two studies conducted to understand and exploit sentiments in software engineering activities. To address sub-problem III (related to developers' copy-paste actions), we present three studies in subsection 3.2.1, subsection 3.2.2, and subsection 3.2.3 respectively. To address sub-problem IV, we present the study in subsection 3.3. Finally, we summarize the chapter in Section 3.4.

3.1 Mining and Analyzing Developers' Sentiments

In this section, first, we describe the work performed to advance state-of-the-art tools for sentiment analysis in software engineering. Then, we describe the empirical studies where we mine and analyze sentiments expressed in software engineering textual artifacts.

3.1.1 Improving Sentiment Analysis in Software Engineering Text

Problem Background: Here we address the sub-problem I that we have described along with its background in subsection 1.1.1. Recalling, *the software engineering community demands a more accurate and automatic sentiment analysis tool.*

Objective: To address the sub-problem I, first, we aim to identify the difficulties responsible for low accuracy of general purpose sentiment tools when applied in the software engineering domain.

Then, the identified difficulties are carefully addressed to develop a tool for improved sentiment analysis in text especially designed for application in the software engineering domain.

Methodology: Using a benchmark dataset, we carry out an in-depth exploratory study and expose 12 difficulties in automatic sentiment analysis in textual content in software engineering. To overcome the majority of those difficulties, we develop a domain dictionary specific for software engineering text. We also develop a number of heuristics to address some of the other identified difficulties. Our new domain dictionary and the heuristics are integrated into `SentiStrength-SE`, a tool we develop for improved sentiment analysis in textual contents in a technical domain, especially in software engineering.

Results: Over a large dataset consisting of 5,600 issue comments, we carry out quantitative comparisons of our domain-specific `SentiStrength-SE` with the three most popular domain independent tools/toolkits (i.e., `NLTK` [46], `Stanford NLP` [47], and the `SentiStrength` [45]. The empirical comparisons suggest that our domain-specific `SentiStrength-SE` is *significantly superior* to its domain independent counterparts in detecting sentiments in software engineering textual artifacts.

3.1.2 A Comparison of Domain Specific Sentiment Analysis Tools

Problem Background: Sentiment Analysis (SA) in software engineering (SE) text has drawn immense interests recently. The poor performance of general-purpose SA tools, when operated on SE text, has led to the recent emergence of domain-specific SA tools specially designed for SE text. However, these domain-specific tools were tested on a single dataset and their performances were compared mainly against general-purpose tools. *Thus, two things remain unclear*: (i) *how well these tools really work on other datasets*, and (ii) *which tool to choose in which context*.

Objective: We aim to answer two following questions: (i) Can we identify a tool, which shows the highest accuracy across different datasets? (ii) To what extent do the different sentiment analysis tools (dis)agree with each other?

Methodology: We select two machine learning based tools `Senti4SD` [55] and `EmoTxt` [56] and a rule based tool `SentiStrength-SE` [109, 110], that are designed to detect sentiments in software engineering texts. Then, we operate those three tools on three ground-truth datasets collected from the JIRA issue comments [52], Stack Overflow posts [55] and code review comments [111]. For each tool, the accuracy of sentiment detection is measured in terms of *precision*, *recall*, and *F-score* separately computed for each of the three sentimental polarities (i.e., positivity, negativity and neutrality). We also compute agreements among the tools for each sentiment.

Results: Our study reveals that the individual tools exhibit their best performance on the dataset they were originally tested at the time of their release. The overall accuracies of the tools tend to decrease when they are operated on a different dataset. The accuracies of the tools largely vary across different datasets and sentimental polarities. Thus, *none* of the tools demonstrates *substantially* superior accuracies across sentiments and datasets. However, `SentiStrength-SE` is found to have consistently exhibited the highest recall and F-score while maintaining competitive precision across all the datasets and sentiments.

From agreement analysis among the tools, we find that the tools' agreements largely vary (between 49.74% and 94.70%) depending on the datasets they are operated on and the sentimental polarities they detect. The tools' agreements remain the lowest in the detection of negative sentiments. Their accuracies also remain lower in the detection of negative sentiments compared to non-negative sentiments. Thus, we suspect that all the tools more or less struggle in accurately detecting negative sentiments in SE text.

3.1.3 A Comparison of Dictionary Building Methods for Sentiment Analysis

Problem Background: Sentiment Analysis (SA) in Software Engineering (SE) texts suffers from low accuracy primarily due to the lack of an effective dictionary. The use of a *domain-specific* dictionary can improve the accuracy of SA in a particular domain. To build a dictionary, several approaches [112, 113, 114, 115] have been attempted in the past resulting in variations in their performances [114, 116, 115]. The construction of a domain dictionary is a tedious task [115] and it is often difficult to know in advance which approach for dictionary building can suite better for a particular domain [113]. *A few studies were conducted on texts from non-technical domains (e.g., movie and restaurant reviews) to find out appropriate dictionaries for those domains* [114, 117]. *However, no such study was performed for a technical domain such as SE.*

Objective: To identify the dictionary building methods, which have higher/lower potential to perform well in constructing a domain dictionary for SA in SE texts.

Methodology: In this study, we use our recently developed SA tool `SentiStrength-SE` [110]. This lexical tool is developed in a modular manner to allow replacing its default dictionary with a different one. we study four different dictionaries (`SentiStrength` [45], `AFINN` [118], `MPQA` [119], and `VADER` [120]). We select four dictionaries based on their well-establishment in SA and recency of their development. In addition, we make sure that those selected dictionaries are developed using distinct methodologies. We use a benchmark dataset [52], which is the only publicly available such dataset in the SE domain [110, 52]. In our work, we use the Group-2 and Group-3 portions of the dataset [52] containing 1,600 and 4,000 issue comments extracted from the JIRA issue tracking system. For each of the four dictionaries, we configure `SentiStrength-SE` to use that particular dictionary and run the tool on issue comments. The outputs of the tool are compared with the annotated benchmark dataset to compute the performance of each dictionary in terms of precision, recall (and F-score) in the detection of positive, negative, and neutral sentiments.

Results: Considering the overall average accuracy, the SentiStrength dictionary appears to have performed the best, followed by AFINN and VADER. Although AFINN performs slightly better than SentiStrength in detecting positive and neutral sentiments, its performance is much worse compared to SentiStrength in detection of negative sentiments. Thus, AFINN falls behind the SentiStrength dictionary in overall accuracy.

It is interesting that MPQA is found to have performed the worst, although the dictionary incorporates contextual information and subjectivity. SE text being informal, containing technical jargons, often including misspelled words and grammatically incorrect sentences can be among the reasons for MPQA's poor performance in our study. Thus, it appears that simple lexicon-based approaches for dictionary creation work better for SA in SE text.

3.1.4 Detection of Emotions in the Valence-Arousal Space

Problem Background: Software engineering specific sentiment analysis tools are limited in capturing sentiments at the necessary depth [50]. *Existing approaches are able to detect* valence *(i.e., positivity and negativity of emotional polarities) only and fail to capture* arousal *or specific sentimental/emotional states such as* excitement, stress, depression, *and* relaxation. At work, software developers frequently experience these sentiments [31], which can be attributed to their work progress. For example, a developer typically feels *relaxed*, if he makes enough progress in his assigned jobs. Otherwise, the developer feels *stressed. Thus, these sentiments need to be identified* [121] *with improved accuracy where the existing approaches fall short* [50].

Objective: We develop an emotion analysis tool DEVA, which is capable of capturing the aforementioned sentimental/emotional states through the detection of both arousal and valence in software engineering text.

Methodology: DEVA applies a dictionary-based lexical approach particularly designed for operation on software engineering text. For the capturing both arousal and valence, the tool uses two separate dictionaries (an arousal dictionary and a valence dictionary) that we develop by exploiting a general-purpose dictionary and two domain dictionaries especially crafted for software engineering text. DEVA also includes a preprocessing phase and seven heuristics to improve the accuracy of detection of those sentiments.

Results: For empirical evaluation of DEVA, we construct a ground-truth dataset consisting of 1,795 *JIRA* issue comments, each of which are manually annotated by three human raters. *This dataset is also a significant contribution to the community*. From a quantitative evaluation using this dataset, DEVA is found to have achieved 82.19% precision and 78.70% recall. We also implement a baseline approach and compare that against DEVA. A recently released similar, (but not identical) tool TensiStrength [122] is also compared against our DEVA. From the comparisons, DEVA is found substantially superior to both the baseline and TensiStrength.

3.1.5 Improving Detection of Emotions in the Valence-Arousal Space

Problem Background: While the dictionary and rule based emotion mining tool `DEVA` performs better compared to its counterparts, it comes with a few limitations, such as out of vocabulary word and unseen rules, which are specific to dictionary and rule based tools. Such limitation can be overcome using Machine Learning based techniques.

Objective: We develop the first *Machine Learning (ML) based* tool `MarValous` for improved detection of four emotional states *excitation, stress, depression,* and *relaxation* expressed in software engineering texts.

Methodology: For improved classification performances, `MarValous` consists of two major modules: (i) data preprocessing and (ii) feature selection. In the preprocessing phase, we sanitize the input text to get rid of probable noises (e.g., code snippets, URL, and numeric expressions), which otherwise could mislead classification. From the sanitized texts, we select seven features that include (i) n-gram, (ii) emoticons, (iii) interjections, (iv) exclamation mark, (v) uppercase words (e.g., GOOD), (vi) elongated words (e.g., goood), and (vii) use of $+1$ and -1 in sentences (it is a common practice of developers to put $+1$ and -1 to express positive and negative emotions in comments while discussing technical issues among them in Stack Overflow and JIRA).

Then, we select following nine ML algorithms based on their popularity in sentiment/emotion classification.

(i) Adaptive Boosting (AB) [105], (ii) Decision Tree (DT) [105, 123], (iii) Gradient Boosting Tree (GBT) [124], (iv) K-nearest Neighbors (KNN) [105], (v) Naive Bayes (NB) [105, 125], (vi) Random Forest (RF) [126], (vii) Multilayer Perceptron (MLP) [127], (viii) Support Vector Machine with Stochastic Gradient Descent [123] (SGD), and (ix) Linear Support Vector Machine (SVM) [105, 128].

Results: To evaluate the performance of `MafrValous`, we create a unifying dataset by combining two different benchmark datasets created by Islam and Zibran [129] and Novielli et al. [130]. To identify the best ML algorithm with the best features' combination, we run each of the algorithm using different combinations of the features on the dataset. From quantitative evaluations, we find that the algorithm SVM with all the seven features performs the best. Then, we have released `MarValous` by setting SVM as its default algorithm while enabling all those features in it.

Then, we compare the performance of `MarValous` against the only available baseline tool `DEVA`. From a quantitative evaluation we find that, on average, across all the emotions, `MarValous` outperforms the baseline, as it has achieved 19.04% higher precision and 8.19% higher recall values compared to DEVA.

3.1.6 Understanding and Exploiting Developers' Sentimental Variations

Problem Background: We address the sub-problem II described along with its background in subsection 1.1.3. Recalling, *here the software engineering community demands more empirical studies to understand the roles and impacts of sentiments in the software engineering domain*.

Objective: To address sub-problem II, we aim to conduct a quantitative empirical study on the characteristics and impacts of sentiments extracted from developers' commit messages.

Methodology: To meet the objective, we ask the following five research questions to answer. (i) RQ1: do developers express different levels (e.g., high, low) and polarity (i.e., positivity, negativity, and neutrality) of sentiments when they commit different types (e.g., bug-fixing, new feature implementation, refactoring, and dealing with energy and security-related concerns) of development tasks? (ii) RQ2: can a group of developers be distinguished who express more sentiments (positive or negative) in committing a particular type (e.g., bug-fixing) of tasks? (iii) RQ3: do the developers' polarity (i.e., positivity, negativity, and neutrality) of sentiments vary in different days of a week and in different times of a day? (iv) RQ4: do the developers' sentiments have any impact on the lengths of commit comments they write? and (v) RQ5: do the sizes of the projects and development teams have any impacts on the developers' emotional states at work?

We study commit messages for open-source projects obtained through `Boa` [131]. Boa is a recently introduced infrastructure with a domain specific language and public APIs to facilitate mining software repositories. We use the largest (as of February 2016) dataset from `Boa`, which is categorized as “full (100%)" and consists of more than 7.8 million projects collected from `GitHub` before September 2015.

From this large dataset, we select the top 50 projects having the highest number of commits. We study all the commit messages in these projects, which constitute 490,659 commit comments. Associated information such as committers, commit timestamps, types of underlying work, revisions and project IDs are kept in a local relational database for convenient access and query.

For each of the commit messages, we compute the sentimental scores. Then, to answer a research question, we identify the required attributes from the collected dataset, and then, conduct various statistical analyses between the identified attributes and computed emotional scores. For example, to answer RQ1, we identify types of tasks of each of the commit messages and calculate a task-type wise emotional score. Then, we conduct *Mann-Whitney-Wilcoxon (MWW)* statistical test [132] to verify whether developers express different levels of sentiments when they commit different types of tasks.

Results: It is found that the polarities of the developers' sentiments significantly vary depending on the type of tasks they are engaged in. The developers express equally high positive and negative sentiments in committing in energy-aware tasks compared to other tasks. With respect to the polarities of commit messages, positive sentiments are found to be significantly higher than negative sentiments in commits for bug-fixing and refactoring tasks. On the other hand, negative sentiments are significantly higher in security-related commits. Surprisingly, the same scenario is found for new feature implementation commits.

A significant positive correlation is found between the lengths of commit messages and the sentiments expressed in developers. When the developers remain emotionally active, they tend to write longer commit comments. The developers tend to render in them more positive sentiments when they work in smaller projects or in smaller development teams, although the difference is not very large. Surprisingly, no significant variations are found in the developers' sentiments in commit messages posted at different times and days of a week.

Based on emotional contents in commit messages, a group of developers can be distinguished who express more positive sentiments at bug-fixing commit messages, another group with the opposite trait, and the third group of developers who equally render both positive and negative sentiments at bug-fixing activities. The same approach can be applied for other types of tasks to distinguish potential developers for improved tasks assignment.

3.1.7 Sentiment Analysis of Software Bug Related Commit Messages

Problem Background: Few studies have *successfully* used sentiment as a factor in prioritizing applications' features to develop [133] and in predicting qualities of developers' interactions (e.g., asking questions and answering) in technical forums [134, 135, 136] and bug severity [137]. Considering the above studies, it deems sentiments can be an influential factor to be used in complex machine learning and deep learning systems to predict bugs (i.e., buggy commits) in software. However, before using sentiments to predict bugs, we need to empirically evaluate of such possibility in the context.

Objective: To address sub-problem II, here we study the variations of polarities (i.e., positivity, negativity, and neutrality) of sentiments expressed in two types of commit messages, (i) *bug-introducing* and (ii) *bug-fixing*, which are posted by developers contributing to open-source projects.

Methodology: In particular, we ask and answer the following two research questions- (i) RQ1: do developers express different levels (e.g., high, low) and polarity (i.e., positivity, negativity, and neutrality) of sentiments in bug-introducing and bug-fixing commits? (ii) RQ2: do the developers? polarity (i.e., positivity, negativity, and neutrality) of sentiments vary in different times of a day in bug-introducing and bug-fixing commits.

To address the aforementioned research questions, we extract bug-introducing and bug-fixing commit messages from three selected projects. To study the relationship between developers sentiments and times of a day when commit comments are posted (i.e., RQ2), we divide the 24 hours of a day in three periods (a) 00 to 08 hours as *before working hours*, (b) 09 to 17 hours as regular *working hours* and (c) 18 to 23 hours as *after working hours*. Then, for each project, we organize the commit messages into three disjoint sets based on their timestamps of posting. Then, we compute emotional scores of commit messages using the tool `SentiStrength-SE` [110, 109], which is the first domain specific sentiment analysis tool for software engineering texts. Finally, to answer the research questions we conduct statistical analyses.

Results: In our study, we find that both bug-introducing and bug-fixing commit messages have overall statistically significantly higher positive emotional scores compared to negative emotional scores. We also observe similar findings while analyzing emotional scores in the bug-introducing and bug-fixing commit messages with respect to three working periods. An exception is found in the latter case where no significant difference found between positive and negative emotional score in bug-fixing commit messages posted during *after work hours*.

While comparing with respect to a particular sentiment (i.e., for positive or negative sentiment), we find no significant difference between overall emotional scores in bug-introducing and bug-fixing commit messages. The earlier finding also holds true while analyzing emotional scores in bug-introducing and bug-fixing commit messages with respect to three working periods with an exception where positive emotional scores are significantly higher in bug-fixing commits posted during *working hours*.

3.2 Mining and Analyzing Impacts of Developers' Copy-Paste Actions

Here, we address the sub-problem III described along with its background in subsection 1.2.1. Using state-of-the-art static analysis tools, we mine *code smells* and *security vulnerabilities* in cloned and non-clone code, and compute various metrics (e.g., densities of those *code smells* and *security vulnerabilities* in terms of line and block) to compare and analyze them in different types of clones and non-clone code. We also present another comparative study on the characteristics of the buggy and non-buggy clones from a code quality perspective. *In the following subsection 3.2.1, subsection 3.2.2, and subsection 3.2.3, we present the studies conducted to address sub-problem III.*

3.2.1 A Study on Code Smells in Categories of Clones and Non-Cloned Code

Objective: To address sub-problem III, we conduct a study to identify relationships of *code smells* with different types of code clones and non-clone code.

Methodology: Using a clone detection tool `NiCad`, we detect different types of code clones in 97 software systems. For the detection of vulnerabilities in source code, we use `PMD` (version 5.3.2) [12], which applies a static rule-based approach for source code analysis and identification of potential vulnerabilities in a software system. Upon detection of the clones and code smells, for each of the subject systems, we identify the co-locations of code clones and code smells, distinguish the code smells located in non-cloned portion of code, and compute densities of code smells in cloned and non-cloned code with respect to (w.r.t.) syntactic blocks of code (BOC) as well as w.r.t. lines of code (LOC).

Results: We have found no significant differences between the densities of vulnerabilities in code clones and clone-free source code. Surprisingly, among the three categories (i.e., *Type-1*, *pure Type-2*, and *pure Type-3*) of clones studied in our work, *Type-1* clones are found to be the most vulnerable whereas *pure Type-3* are the least. In addition, our study identifies a set of five vulnerabilities that appear more frequently in cloned code compared to non-cloned code. Another set of 11 vulnerabilities are also distinguished, which are more frequently found in non-cloned code as opposed to cloned code.

3.2.2 A Study on Security Vulnerabilities in Clones and Non-Cloned Code

Objective: To address sub-problem III, we conduct another comparative study on security vulnerabilities and their severities in different types of clones compared to non-clone code.

Methodology: Using a state-of-the-art clone detector and two reputed security vulnerability detection tools, we detect clones and vulnerabilities in 8.7 million lines of code over 34 software systems. After detecting clones and vulnerabilities in the software systems, we determine locations of the detected vulnerabilities in different types of code (i.e., cloned and non-cloned code). A vulnerability is said to be located in cloned code if the reported source code line number of that vulnerability included in a cloned block, otherwise, the vulnerability is located in the non-cloned block. For each of the subject systems, we identify the co-locations of code clones and vulnerabilities, distinguish the vulnerabilities located in a non-cloned portion of code. Then, we perform a comparative study of the vulnerabilities identified in different types of clones and non-cloned code.

Results: Our study reveals that the security vulnerabilities found in code clones have higher severity of security risks compared to those in non-cloned code. However, the proportion (i.e., density) of vulnerabilities in clones and the non-cloned code does not have any significant difference.

3.2.3 On the Characteristics of Buggy Code Clones: A Code Quality Perspective

Objective: To address sub-problem III, we again conduct a comparative study to analyze characteristics of the buggy and non-buggy clones from a code quality perspective.

Methodology: We select 2,077 bug-fixing revisions of three open-source software systems written in Java. We use the `NiCad` [138] clone detector (version 3.5), to detect method/function clones having at least five lines of code in those selected revisions. Clones that are affected by the bug-fixing commits are identified as buggy clones while the rest other clones are considered non-buggy.

Using a free version of `SourceMeter` [139] (version 8.2.0-x64-linux), we compute 29 source code metrics for each of the buggy and non-buggy clones. We use total 29 source code quality metrics grouped into four categories- (i) *complexity metrics*, which measure the complexity of source code elements; (ii) *size metrics*, which measure the basic properties of the analyzed system in terms of different cardinalities (e.g., number of code lines). (iii) *documentation metrics*, which measure the number of comments and documentation of source code elements in the system; and (iv) *coupling metrics*, which measure the inter-dependencies of source code elements. The, we statistically analyze the code quality metrics' values in buggy and non-buggy cloned code.

Results: In our study, we have found that buggy clones have higher complexity and lower maintainability compared to non-buggy cloned methods. Moreover, buggy clones are found to be larger in size measured in terms of the number of statements and lines of code. Surprisingly, we have found that the non-buggy method clones have a higher number of parameters while too many parameters in functions are generally considered problematic and recognized as a code smell [74].

As expected, compared to non-buggy clones, documentation quality of buggy clones are found inferior. Overall, buggy clones are found to be more coupled than non-buggy clones. However, it is interesting to have found that the buggy cloned methods have significantly higher outgoing dependencies (i.e., outgoing invocations) compared to non-buggy clones. In the case of incoming dependencies, no significant difference is found between buggy and non-buggy clones.

3.3 Mining and Analyzing Fixing Patterns of Developers' Mistakes

Objective: We address the sub-problem IV described along with its background in subsection 1.3.1. In this work, we capture both bug-fixing *edit* patterns and *nesting* patterns (i.e., frequent nested code structures) of bug-fixing edits through an in-depth (quantitative and qualitative) analysis of 4,653 buggy revisions of five software systems drawn from diverse application domains.

Methodology: For each subject system, we collect the bug-fixing revisions. Then, for each bug-fixing revision, using AST based code differencing tools, we detect differences between the bug-fixing revision and its immediate previous revision. Collections of such AST differences are then analyzed to detect bug-fixing edit patterns and dominant nested code structures of code changes to fix bugs.

Results: In this work, we have reported 38 bug-fixing *edit* patterns, which is the highest number of bug-fixing *edit* patterns identified in a single study. Moreover, we have discovered four new bug-fixing *edit* patterns. The rest 34 identified bug-fixing *edit* patterns confirm those reported earlier.

Using sequential pattern mining and clustering techniques, we have also exposed 37 new bug-fixing *nesting* patterns, which capture the locations of the bug-fixing edits within the nested code structure surrounding them. These new set of *nesting* patterns is a novel contribution that adds a new dimension to our understanding of bug-fixing patterns.

Our analysis of the *nesting* patterns reveals additional insights into bug-fix patterns. We have found that any nodes/blocks associated with `if` blocks are the most bug-prone. The *nesting* pattern "`if` block inside `loop` block" experience the highest number bug-fixing edits, followed by the "`if` block inside another `if` block" *nesting* pattern. Our analysis of the nesting patterns also indicates nodes/blocks inside `try-catch` and `synchronized` are bug-prone.

3.4 Summary

In this chapter, we have presented our research methodologies that are used to address sub-problems I, II, III and IV. To address sub-problem I, we have developed two state-of-the-art tools (i.e., `SentiStrenghth-SE` and `DEVA`) to detect sentiments/emotions at different levels. We have also compared our tool `SentiStrenghth-SE` against its domain-specific counterparts and found that `SentiStrenghth-SE` shows consistently higher recall value compare to other tools. However, our `DEVA` is the only tool available to detect *excitement*, *stress*, *depression*, and *relaxation* in software engineering textual artifacts. We have also compared four dictionaries, developed using four distinct methods, to compare their sentiment detection performances in software engineering textual artifacts.

Then, we have conducted a quantitative empirical study of the emotional variations in different types of development activities (e.g., bug-fixing tasks), development periods (i.e., days and times) and in projects of different sizes involving teams of variant sizes. The study also includes an in-depth investigation of emotions' impacts on software artifacts (i.e., commit messages) and exploration of scopes for exploiting emotional variations in software engineering activities. Next, we also have studied the emotional variations in bug-introducing and bug-fixing commit messages.

Later, we have presented a comparative study on different types of clones and non-cloned code on the basis of their *code smell*. The study has found no significant differences between the densities of vulnerabilities in code clones and clone-free source code. Then, another similar comparative study is conducted on the basis of security vulnerabilities instead of code smells. The latter study has revealed that the security vulnerabilities found in code clones have higher severity of security risks compared to those in non-cloned code. Then, we have presented one more comparative study on the characteristics of the buggy and non-buggy clones from a code quality perspective. Finally, we have presented an in-depth study of both bug-fixing *edit* patterns and *nesting* patterns (i.e., frequent nested code structures) of bug-fixing edits.

Chapter 4

Detecting Developers' Sentiments

Accurate detection of sentiments/emotions with high accuracy is the key to conduct sentiments/emotions related analysis in software engineering. However, sentiment analysis in software engineering textual artifacts has long been suffering from inaccuracies in those few tools available for the purpose. We conduct an in-depth qualitative study to identify the difficulties responsible for such low accuracy. Majority of the exposed difficulties are then carefully addressed through building a domain dictionary and appropriate heuristics. These domain-specific techniques are then realized in `SentiStrength-SE`, a tool we have developed for improved sentiment detection in text especially designed for application in the software engineering domain.

This chapter is organized as follows. In Section 4.1, we provide the motivation behind this particular work. In Section 4.2, we describe the exploratory study to identify the difficulties in detecting sentiments. Section 4.3 describes how we develop a tool for improved sentiment analysis by overcoming most of the difficulties. Section 4.4 illustrates how we evaluate our sentiment analysis tool. In Section 4.5, we identify the limitations of the tool and future possibilities. We describe the related work in Section 4.6. Finally, we conclude in Section 4.7.

4.1 Introduction

Emotions are an inseparable part of human nature, which influence people's activities and interactions, and thus emotions affect task quality, productivity, creativity, group rapport and job satisfaction [26]. Software development, being highly dependent on human efforts and interactions, is more susceptible to emotions of the practitioners. Hence, a good understanding of the developers' emotions and their influencing factors can be exploited for effective collaborations, task assignments [29], and in devising measures to boost up job satisfaction, which, in turn, can result in increased productivity and projects' success.

Several studies have been performed in the past for understanding the role of human aspects on software development and engineering. Some of those earlier studies address *when* and *why* employees get affected by emotions [26, 34, 4, 41, 43], whereas some other work address *how* [60, 37, 38, 62, 39, 63, 31, 140] the emotions impact the employees' performance at work.

Attempts are made to capture the developers' emotions in the workplace by means of traditional approaches such as, interviews, surveys [31], and biometric measurements [32]. Capturing emotions with the traditional approaches is more challenging for projects relying on geographically distributed team settings and voluntary contributions (e.g., open-source projects) [34, 33]. Moreover, the traditional approaches involving direct observations and interactions with the developers often hinder their natural workflow. Thus, to supplement or complement those traditional approaches, recent attempts detect sentiments from the software engineering textual artifacts such as issue comments [34, 41, 37, 38, 39, 40, 35, 36], email contents [43, 42], and forum posts [4, 44].

For automated extraction of sentiments from textual artifacts in the software engineering domain, three tools (i.e., `SentiStrength` [45], `NLTK` (Natural Language Toolkit) [46], and `Stanford NLP` [47]) are used while the use of `SentiStrength` is found dominant [48, 49]. However, software engineering studies [41, 43, 37, 35, 36, 48, 50, 51] involving sentiment analysis repeatedly report concerns about the accuracy of those sentiment analysis tools in the detection of sentimental polarities (i.e., negativity, positivity, and neutrality) of plain text contents. For example, when applied in the software engineering domain, `SentiStrength` and `NLTK` are respectively reported to have only 29.56% and 52.17% precision in identifying positive sentiments, and even lower precision of 13.18% and 23.45% respectively in the detection of negative sentiments [43, 48].

Those sentiment analysis tools are developed and trained using data from non-technical social networking media (e.g., twitter posts, forum posts, movie reviews) and when operated in a technical domain such as software engineering, their accuracy substantially degrades largely due to domain-specific variations in meanings of frequently used technical terms. Although such a domain dependency is indicated as a general difficulty against automated sentiment analysis in textual content, we need a deeper understanding of why and how such domain dependencies affect the performance of the tools, and how we can mitigate them. Indeed, the software engineering community demands a more accurate automatic sentiment analysis tool [41, 43, 38, 35, 36, 50, 52, 53]. In this regard, this chapter makes three major contributions:

- Using a large benchmark dataset, we carry out an in-depth exploratory study for exposing the difficulties in automatic sentiment analysis in textual content in a technical domain such as software engineering.

- We develop a *domain dictionary* specific for software engineering text. To the best of our knowledge, this is the first domain-specific sentiment analysis dictionary for the software engineering domain.

- We propose techniques and realize those in `SentiStrength-SE`, a prototype tool that we develop for improved sentiment analysis in software engineering textual content. The tool is also made freely available online [54]. `SentiStrength-SE` is the first domain-specific sentiment analysis tool especially designed for software engineering text.

Instead of building a tool from scratch, we develop our `SentiStrength-SE` on top of `SentiStrength` [45], which, till date, is the most widely used tool for automated sentiment analysis in software engineering [110]. From quantitative comparison with the original `SentiStrength` [45], `NLTK` and `Stanford NLP` as operated in the software engineering domain, we find that our domain-specific `SentiStrength-SE` significantly outperforms those domain independent tools/toolkits. We also separately evaluate the contributions of individual major components (i.e., the domain dictionary and heuristics) of our `SentiStrength-SE` in sentiment analysis in software engineering text. Our evaluations demonstrate that, for software engineering text, domain-specific sentiment analysis techniques perform substantially better in detecting sentiments accurately. We further conduct a qualitative evaluation of our tool. Based on the exploratory study and the qualitative evaluation, we outline plans for further improvements in automated sentiment analysis in the software engineering area.

This chapter is a significant extension to our recent work [110]. This chapter presents *new evidence and insights* by including a *deeper analysis* of the difficulties in automated sentiment analysis in software engineering text. The techniques applied in the development of `SentiStrength-SE` are described in *greater detail*. The empirical evaluation of the tool is substantially extended with *deeper qualitative analyses and direct comparisons* with `NLTK` and `Stanford NLP` in addition to the previously published comparison with the original `SentiStrength`. We include separate evaluations of the individual major components (i.e., the domain dictionary and heuristics) of `SentiStrength-SE`. The quantitative comparisons are validated in the light of statistical tests of significance.

4.2 Exploratory Study of the Difficulties in Sentiment Analysis

To explore the difficulties in automated sentiment detection in text, we conduct our qualitative analysis around the *Java version* of `SentiStrength` [45]. This *Java version* is the latest release of `SentiStrength`, while the older version, strictly for use on Windows platform, is still available. As mentioned before, `SentiStrength` is a state-of-the-art sentiment analysis tool most widely adopted in the software engineering community. The reasons for choosing this particular tool are further justified in Section 4.6.

English dictionaries consider the words 'emotion' and 'sentiment' as synonymous, and accordingly the words are often used in practice. Although there is arguably a subtle difference between the two, in describing this work, we consider them synonymous. We formalize that, aside from subjectivity, a human expression can have two perceivable dimensions: sentimental *polarity* and sentimental *intensity*. Sentimental *polarity* indicates the positivity, negativity, or neutrality of expression while sentimental *intensity* captures the strength of the emotional/sentimental expression, which sentiment analysis tools often report in numeric emotional scores.

4.2.1 Benchmark Data

In our work, we use a “Gold Standard" dataset [52, 141], which consists of 5,992 issue comments extracted from JIRA issue tracking system. The entire dataset is divided in three groups named as Group-1, Group-2 and Group-3 containing 392, 1,600 and 4,000 issue comments respectively. Each of the 5,992 issue comments are manually interpreted by n distinct human raters [52] and annotated with emotional expressions as found in those comments. For Group-1, $n = 4$ while for Group-2 and Group-3, $n = 3$. This is the only publicly available such dataset in the software engineering domain [52, 110].

A closed set $\mathcal{E}$ of *emotional expressions* are used in the annotation of the issue comments in the dataset, where $\mathcal{E}$ ={*joy, love, surprise, anger, sad, fear*}. The human raters labeled each of the issue comments depending on whether or not they found the sentimental expressions in the comments. Formally,

$$\mathcal{F}^{r_j}_{\mathcal{E}_i}(C) = \begin{cases} 1, & \text{if emotion } \mathcal{E}_i \text{ is found in } C \text{ by rater } r_j. \\ 0, & \text{otherwise.} \end{cases}$$

An example of human annotations of an issue comment from the dataset is shown in Table 4.1.

Table 4.1: Example of annotation of an issue comment by four human raters

Issue comment (Comment ID-53257): `Thanks for the patch; Michale. Applied with a few modifications.`

Human	**Emotions ($\mathcal{E}_i$)**					
Raters (r_j)	**Joy**	**Love**	**Surprise**	**Anger**	**Sadness**	**Fear**
Rater-1 (r_1)	**1**	**1**	0	0	0	0
Rater-2 (r_2)	0	0	0	0	0	0
Rater-3 (r_3)	**1**	0	0	0	0	0
Rater-4 (r_4)	**1**	0	0	0	0	0

Interpretation: rater-1 found ‘joy’ and ‘love’ in the comment, while rater-3 and rater-4 found the presence of only ‘love’ but rater-2 did not identify any of the emotional expressions.

4.2.2 Emotional Expressions to Sentimental Polarities

Emotional expressions joy and $love$ convey *positive* sentimental polarity, while $anger$, $sadness$, and $fear$ express *negative* polarity. In some cases, an expression of $surprise$ can be positive in polarity, denoted as $surprise^+$, while other cases can convey a *negative surprise*, denoted as $surprise^-$. Thus the issue comments in the benchmark dataset, which are annotated with *surprise* expression, need to be further distinguished based on the sentimental polarities they convey. Hence, we get each of such comments reinterpreted by three additional human (computer science graduate students) raters, who independently determine polarities of the surprise expressions in each comments.

We consider a *surprise* expression in a comment polarized negatively (or positively), if two of the three rates identify negative (or positive) polarity in it. We found 79 issue comments in the benchmark dataset, which were annotated with the *surprise* expression. 20 of them express *surprise* with positive polarity and the rest 59 convey negative *surprise*.

Then we split the set $\mathcal{E}$ of emotional expressions into two disjoint sets as $\mathcal{E}_+ = \{joy, love, surprise^+\}$ and $\mathcal{E}_- = \{anger, sad, fear, surprise^-\}$. Thus, $\mathcal{E}_+$ contains only the positive sentimental expressions and $\mathcal{E}_-$ contains only the negative sentimental expressions. A similar approach is also used in other studies [48, 142] to categorize emotional expressions according to their polarities.

4.2.3 Computation of emotional scores from human rated dataset

For each of the issue comments in the "Gold Standard" dataset, we compute sentimental polarity using the polarity labels assessed by the human raters. For an issue comment C rated by n human raters, we compute a pair $\langle \rho_c^{r_j}, \eta_c^{r_j} \rangle$ of values for each of the n raters r_j (where $1 \leq j \leq n$) using Equation 4.1 and Equation 4.2:

$$\rho_c^{r_j} = \begin{cases} 1, & \text{if } \sum\limits_{\mathcal{E}_i \epsilon \mathcal{E}_+} \mathcal{F}_{\mathcal{E}_i}^{r_j}(C) > 0 \\ 0, & \text{otherwise.} \end{cases} \tag{4.1}$$

$$\eta_c^{r_j} = \begin{cases} 1, & \text{if } \sum\limits_{\mathcal{E}_i \epsilon \mathcal{E}_-} \mathcal{F}_{\mathcal{E}_i}^{r_j}(C) > 0 \\ 0, & \text{otherwise.} \end{cases} \tag{4.2}$$

Thus, if a rater r_j finds the presence of any of the positive sentimental expressions in the comment C, then $\rho_c^{r_j} = 1$, otherwise $\rho_c^{r_j} = 0$. Similarly, if any of the negative sentimental expressions are found in the comment C, then $\eta_c^{r_j} = 1$, otherwise $\eta_c^{r_j} = 0$.

An issue comment C is considered neutral in sentimental polarity, if we get the pairs $\langle \rho_c^{r_j}, \eta_c^{r_j} \rangle$ for at least $n-1$ (i.e., majority) raters where $\rho_c^{r_j} = 0$ and $\eta_c^{r_j} = 0$. If the comment is not neutral, then we determine the positive and negative sentimental polarities of that issue comment. To do that, using the following equations, we count the number of human raters, $\mathcal{R}_+(C)$ who found positive sentiment in the comment C and also the number of raters, $\mathcal{R}_-(C)$, who found negative sentiment in the comment C.

$$\mathcal{R}_+(C) = \sum_{j=1}^{n} \rho_c^{r_j} \qquad \text{and} \qquad \mathcal{R}_-(C) = \sum_{j=1}^{n} \eta_c^{r_j}$$

An issue comment C is considered exhibiting positive sentiment, if at least $n-1$ human raters found positive sentiment in the message. Similarly, we consider a comment having negative sentiment if at least $n-1$ raters found negative sentiment in it. Finally, we compute the sentimental polarities of an issue comment C as a pair $\langle \rho_c^h, \eta_c^h \rangle$ using Equation 4.3 and Equation 4.4.

$$\rho_c^h = \begin{cases} 0, & \text{if } C \text{ is neutral} \\ +1, & \text{if } \mathcal{R}_+(C) \geq n-1 \\ -1, & \text{otherwise.} \end{cases} \tag{4.3}$$

$$\eta_c^h = \begin{cases} 0, & \text{if } C \text{ is neutral} \\ +1, & \text{if } \mathcal{R}_-(C) \geq n-1 \\ -1, & \text{otherwise.} \end{cases} \tag{4.4}$$

Thus, $\rho_c^h = 1$, only if the comment C has positive sentiment and $\eta_c^h = 1$ only if the comment contains negative sentiment. Note that, a given comment can exhibit both positive and negative sentiments at the same time. A comment is considered sentimentally neutral when the pair $\langle \rho_c^h, \eta_c^h \rangle$ for the comment appear to be $\langle 0, 0 \rangle$. An issue comment is discarded from our study if at least $n-1$ human raters (i.e., majority) could not agree on any particular sentimental polarity of the comment. We have found 33 such comments in Group-2 dataset that are excluded from our study. Similar approach is also followed to determine sentiments of comments in another study [48].

4.2.3.1 Illustrative Example of Computing sentimental Polarity

Consider the issue comment in Table 4.1. For this issue comment, we compute the pair $\langle \rho_c^{r_j}, \eta_c^{r_j} \rangle$ for all four raters (i.e., $n = 4$). As for only one (the second rater) out of four raters we get the pair as $\langle 0, 0 \rangle$, the comment is not considered neutral. Hence, we compute the values of $\mathcal{R}_+(C)$ and $\mathcal{R}_-(C)$, which are three and zero respectively. $\mathcal{R}_+(C)$ being three satisfies the condition of $\mathcal{R}_+(C) \geq n-1$. Thus, $\rho_c^h = 1$, which means that the comment in Table 4.1 has positive sentiment. For the same comment $\mathcal{R}_-(C) < n-1$ and so $\eta_c^h = -1$, which signifies that the comment has no negative sentiment.

4.2.4 Sentiment Detection Using `SentiStrength`

We apply `SentiStrength` to determine the sentiments expressed in the issue comments in Group-1 of the “Gold Standard" dataset. Sentiment analysis using `SentiStrength` on a given piece of text (e.g., an issue comment) C computes a pair $\langle \rho_c, \eta_c \rangle$ of integers, where $+1 \leq \rho_c \leq +5$ and $-5 \leq \eta_c \leq -1$. Here, ρ_c and η_c respectively represent the positive and negative sentimental scores for the given text C. A given text C is considered to have positive sentiment if $\rho_c > +1$. Similarly, a text is held containing negative sentiment when $\eta_c < -1$. Besides, a text is considered sentimentally neutral when the sentimental scores for the text appear to be $\langle 1, -1 \rangle$.

Hence, for the pair $\langle \rho_c, \eta_c \rangle$ of sentimental scores for an issue comment C computed by `SentiStrength`, we compute another pair of integers $\langle \rho_c^t, \eta_c^t \rangle$ as follows:

$$\rho_c^t = \begin{cases} 1, & \text{if } \rho_c > +1. \\ 0, & \text{otherwise.} \end{cases} \quad \text{and} \quad \eta_c^t = \begin{cases} 1, & \text{if } \eta_c < -1. \\ 0, & \text{otherwise.} \end{cases}$$

Here, $\rho_c^t = 1$ signifies that the issue comment C has positive sentiment, and $\eta_c^t = 1$ implies that the issue comment C has negative sentiment.

We apply `SentiStrength` to compute sentimental scores for each of the issue comments in the Group-1 portion of the "Gold Standard" dataset and then for each issue comment C, we compute the pair $\langle \rho_c^t, \eta_c^t \rangle$, which represents the sentimental polarity scores for C.

4.2.5 Analysis and Findings

For each of the 392 issue comments C in Group-1, we compare the sentimental polarity scores $\langle \rho_c^t, \eta_c^t \rangle$ produced from `SentiStrength` and the scores $\langle \rho_c^h, \eta_c^h \rangle$ computed using our approach described in Section 4.2.3. We find a total of 151 comments, for which the $\langle \rho_c^t, \eta_c^t \rangle$ scores obtained from `SentiStrength` do not match with $\langle \rho_c^h, \eta_c^h \rangle$. This implies that for those 151 issue comments `SentiStrength`'s computation of sentiments are probably incorrect.

Upon developing a solid understanding of the sentiment detection algorithm of `SentiStrength`, we then carefully go through all of those 151 issue comments to identify the reasons/difficulties, which mislead `SentiStrength` in its identification of sentiments in textual content. We identify 12 such difficulties. Before discussing the difficulties, we first briefly describe the highlights of `SentiStrength`'s internal working mechanism to develop necessary context and background for the reader.

Table 4.2: The role of the dictionary lists in `SentiStrength`'s computation of sentimental scores in text

Sample sentence	**Sent. Scores**		**Dictionary list in use**	**Explanation**
	ρ_c	η_c		
It's a *good* feature.	2	-1	Sentimental words	The sentimental score of the word *'good'* is pre-assigned to 02; so the sentence is assigned positive score 02.
It's a *very* good feature.	3	-1	Booster words	As booster word *'very'* is used before the sentimental word, the sentence is assigned a positive score 03.
It's *not* a good feature.	1	-1	Negations	Sentimental polarity of the sentimental word is inverted in here due to the use of the negation word *'not'* before sentimental word.
It's a *killer feature*.	2	-1	Phrases	*"killer feature"* is a phrase in the dictionary with positive score 02. Although the word *'kill'* carries negative sentiment, its effect is overridden by the sentimental score of the phrase.

Table 4.3: Frequencies of difficulties misleading sentiment analysis

Difficulties		Frequency (%)	Scope*
D_1 :	Domain-specific meanings of words	123 (60.00)	SEDS
D_2 :	Context-sensitive variations in meanings of words	35 (17.07)	SAG
D_3 :	Misinterpretation of the letter 'X'	12 (05.85)	SEDS
D_4 :	Sentimental words in copy-pasted content (e.g., code)	12 (05.85)	SEDS
D_5 :	Difficulties in dealing with negations	08 (03.90)	SAG
D_6 :	Missing sentimental words in dictionary	02 (00.97)	SAG
D_7 :	Spelling errors mislead sentiment analysis	02 (00.97)	SAG
D_8 :	Repetitive numeric characters considered sentimental	01 (00.49)	SST
D_9 :	Wrong detection of proper nouns	01 (00.49)	SST
D_{10} :	Sentimental words in interrogative sentences	01 (00.49)	SST
D_{11} :	Difficulty in dealing with irony and sarcasm	01 (00.49)	SAG
D_{12} :	Hard to detect subtle expression of sentiments	07 (03.41)	SAG

*Here, SEDS = Software Engineering Domain Specific, SAG = Sentiment Analysis in General, SST = Specific to the SentiStrength Tool.

4.2.5.1 Insights into SentiStrength's Internal Algorithm

SentiStrength is a lexicon-based classifier that also uses additional (non-lexical) linguistic information and rules to detect sentiment in plain text written in English [45]. SentiStrength maintains a dictionary of several lists of words and phrases as its key dictionaries to compute sentiments in texts. Among these lists, the *sentimental words list, list of booster words, list of phrases*, and *list of negations words* play a vital role in the computation of sentiments. The entries in all these lists except the list of negation words are pre-assigned with sentimental scores. The negation words in the fourth list are used to invert the sentimental polarity of a term when the term is located after a negation word in text.

For an input sentence, SentiStrength extracts individual words from the sentence and searches for each of the individual words in the *sentimental words list* to retrieve the corresponding sentimental scores. Similar search is made in the *list of booster words* to strengthen or weaken the sentimental scores. The *list of phrases* is used to distinguish groups of words as commonly used phrases. When such a phrase is identified, the sentimental score of the phrase overrides sentimental scores of the individual words, which constitute the phrase. The examples in Table 4.2 articulate how SentiStrength depends on the dictionary of lists for computing sentimental scores in plain texts.

4.2.5.2 Difficulties in Automated Sentiment Analysis in Software Engineering

Table 4.3 presents the number of times we found SentiStrength being mislead by the 12 difficulties as discovered during manual investigation. It is evident in Table 4.3 that *domain-specific meanings of words* is the most prevalent among all the difficulties that are liable for low accuracy of the lexical approach of SentiStrength. However, not all the difficulties are specific to software engineering domain, rather some difficulties impact sentiment analysis in general (including software engineering) while a few are actually specific limitations of the tool `SentiStrength`. The right-most column in Table 4.3 indicates the scopes of the identified difficulties. We now describe 12 difficulties with illustrative examples.

(D_1) Domain-specific meanings of words: In a technical field, textual artifacts include many technical jargons, which have polarities in terms of dictionary meanings, but do not really express any sentiments in their technical context. For example, the words ‘Super’, ‘Support’, ‘Value’ and ‘Resolve’ are English words with known positive sentiment, whereas ‘Dead’, ‘Block’, ‘Default’, and ‘Error’ are known to have negative sentiment, but none of these words really bear any sentiment in software development artifacts.

As `SentiStrength` was originally developed and trained for non-technical texts written in plain English, it identifies those words as sentimental words, which is incorrect in the context of a technical field such as software engineering. In the following comment from the “Gold Standard" dataset, `SentiStrength` considers ‘Error’ as negative sentimental word and detects ‘Support’ and ‘Refresh’ as positive sentimental words. Thus, it assigns both positive and negative sentimental scores to the comment, although the comment is sentimentally neutral.

```
 "This was probably fixed by WODEN-86 which introduced support for the
curly brace syntax in the http location template.  This JIRA can now be
closed.  This test case is now passing ...  There are now 12 errors reported
for Woden on this test caseregenerated the results in r480113.  I'll have
the W3C reports refreshed."
```
(Comment ID: 18059)

(D_2) Context-sensitive variations in meanings of words: Apart from domain-specific meanings of words, in natural language, some words have multiple meanings depending on the context in which they are used. For example, the word ‘Like’ expresses positive sentiment when it is used in a sentence such as *“I like you"*. On the other hand, that same word expresses no sentiment in the sentence *“I would like to be a sailor, said George Washington"*. Again, `SentiStrength` identifies the word ‘Please’ as positive sentimental word, although we find the word is used as neutral to express request in the training dataset. For example, in the comment below, the word ‘Please’ does not express any emotion.

```
 "Updated in 1.2 branch.  David; please download and try 1.2 beta when it
is released in a week or so.."
```
(Comment ID: 4223)

Again, words that are considered inherently sentimental often do not carry sentiments when used to express possibility and uncertainty. Distinguishing the context-sensitive meanings of such words is a big challenge for automated sentiment analysis in text and the lexical approach of `SentiStrength` also falls short in this regard.

For example, in the following issue comment, the sentimental word ‘Nice’ is used simply to express possibility regarding change of something, but `SentiStrength` incorrectly computes positive sentiment in the message.

> `"The change you want would be nice; but is simply not possible. The form data ... Jakarta FileUpload library."` (Comment ID: 51837)

Similarly, in the comment, the sentimental word ‘Misuse’ is used in a conditional sentence, which does not express any sentiment, but `SentiStrength` interprets otherwise.

> `"Added a couple of small points ... if anyone notices any misuses of the document formatting ..."` (Comment ID: 2463)

(D_3) Misinterpretation of the letter ‘X’: In informal computer mediated chat, the letter ‘X’ is often used to mean an action of ‘Kiss’, which is a positive sentiment, and thus recorded in `SentiStrength`’s dictionary. However, in technical domain, the letter is often used as a wildcard. For example, the sequence ‘1.4.x’ in the following comment is used to indicate a collection of versions/releases.

> `"Integrated in Apache Wicket 1.4.x ..."` (Comment ID: 20748)

Since `SentiStrength` uses dot (.) as a delimiter to split a text into sentences, the ‘x’ is considered a one-word sentence and is misinterpreted to have expressed positive sentiment.

(D_4) Sentimental words in copy-pasted content (e.g., code): At commit, the developers often copy-paste code snippets, stack traces, URLs, and camel-case words (e.g., variable names) in their issue comment. Such copy-pasted contents often include sentimental words in the form of variable names and the like, which do not convey any sentiment of the committer, but `SentiStrength` detects those sentimental words and incorrectly associates those sentiments with the issue comment and the committer. Consider the following issue comment, which includes a copy-pasted stack trace.

> `"... Stack: [main] FATAL ... org.apache.xalan.templates .ElemTemplateElement.resolvePrefixTables ..."` (Comment ID: 9485)

The words ‘Fatal’ and ‘Resolve’ (part of the camel case word ‘resolvePrefixTables’), are positive and negative sentimental words respectively in the dictionary of `SentiStrength's`. Hence, `SentiStrength` detects both positive and negative sentiments in the issue comment, but the stack trace content certainly does not represent the sentiments of the developer/committer.

(D_5) Difficulties in dealing with negations: For automated sentiment detection, it is crucial to identify the presence of any negation term preceding a sentimental word, because the negation terms invert the polarity of the sentimental words. For example, the sentence *“I am not in good mood"* is equivalent to *“I am in bad mood"*. When the negation of the positive word ‘Good’ cannot be identified as equivalent to the negative word ‘Bad’, then detection of sentimental polarity goes wrong. The default configuration of `SentiStrength` enables it to detect negation of a sentimental word *only if* the negation term is placed *immediately before* the sentimental word. In all other cases, `SentiStrength` fails to detect negations correctly and often detects sentiments exactly opposite of what is expressed in the text. During our investigation, we find substantial instances where `SentiStrength` is misled by complex structural variations of negations present in the issue comments.

For example, in the following two comments, `SentiStrength` cannot detect negation, which are used before the word 'Bad' (in first comment) and 'Good' (in second comment) correctly and thus misclassified sentiments of those comments.

> `"I haven't seen any bad behavior. I was using open ssh to test this. I used the .... with open ssh to disconnect;"` (Comment ID: 6688)

> `"3.0.0 has been released; closing ... I didn't change the jute - don't think this is a good idea; esp as also effects the ........ Andrew could you take a look at this one?"` (Comment ID: 1725)

In addition, we find that `SentiStrength` is unable to recognized shortened forms of negations such as, “haven't", “havent", “hasn't", “hasnt", “shouldn't", “shouldnt", and “not" since these terms are not included in the dictionary.

(D$_6$) Missing sentimental words in dictionary: Since the lexical approach of `SentiStrength` is largely dependent on its dictionary of lists of words (as discussed in Section 4.2.5.1), the tool often fails to detect sentiments in some texts when the sentimental words used in the texts are absent in the dictionary. For example, the words 'Apology' and 'Oops' in the following two comments express negative sentiments, but `SentiStrength` cannot detect them since those words are not included in its dictionary.

> `"...This is indeed not an issue. My apologies ..."`
> (Comment ID: 20729)

> `"Oops; issue comment had wrong ticket number in it ..."`
> (Comment ID: 36376)

(D$_7$) Spelling errors mislead sentiment analysis: Misspelled words are common in informal text, and the writer often deliberately misspells words to express intense sentiments. For example, the misspelled word 'Happpy' expresses more happiness than the correctly spelled word 'Happy'. Although `SentiStrength` can detect some of such intensified sentiments from such misspelled sentimental words, its ability is limited to only those intentional spelling errors where repetition of certain letters occur in a sentimental word. Most other types (unintentional) of misspelling of sentimental words cause `SentiStrength` fail to find those words in its dictionary and consequently lead to incorrect computation of sentiments. For example, the word ‘Unfortunately’ was misspelled as ‘Unforunatly’ in an issue comment (comment ID: 11978) and “I’ll" was written as ‘ill’ in another (comment ID: 927). `SentiStrength`’s detection sentiments in both of these comments are found incorrect.

(D_8) Repetitive characters considered sentimental: As described before, `SentiStrength` detects higher intensity of sentiments by considering deliberately misspelled sentimental word with repetitive letters. The tool also uses the same strategy for the same purpose by taking into account repetitive characters intentionally typed in words that are not necessarily sentimental by themselves. If anybody writes *“I am goooing to watch movie"* instead of *“I am going to watch movie"*, then the former sentence is considered positively sentimental due to emphasis on the word ‘Going’ by repetition of the letter ‘O’ for three times.

However, this strategy also misguides `SentiStrength` in dealing with some numeric values. For example, in the following comment, `SentiStrength` incorrectly identifies the number ‘20001113’ as a positive sentimental word encountering repetition of the digits ‘0’ and ‘1’.

```
 "See bug 5694 for the ...  20001113 /introduction.html ...  Zip file
with test case (java source and XML docs) 1.  Do you use deferred DOM?
2.  Can you try to run it against Xerces2 beta4 (or the latest code in
CVS?) 3.  Can you provide a sample file?  Thank you."
```
(Comment ID: 6447)

(D_9) Incorrect detection of proper nouns: A proper noun can rightly be considered neutral in sentiment. `SentiStrength` detects a word starting with a capital letter as a proper noun, when the word is located in the middle or end of a sentence. Unfortunately, grammar rules are often ignored in informal text and thus, sentimental words placed in the middle or end of a sentence often end up starting with a capital letter, which cause `SentiStrength` mistakenly disregard the sentiments in those sentimental words. The following issue comment is an example of such a case, where the sentimental word ‘Sorry’ starting with a capital letter is placed in the middle of the sentence and `SentiStrength` erroneously considers ‘Sorry’ as a neutral proper noun.

```
 "Cool.  Thanks for considering my bug report!  ...  About the title of
the bug; in the description; I put:  Sorry for the vague ticket title.  I
don't want to make presumptions about the issue ...  work for passwords."
```
(Comment ID: 76385)

However, the older *Windows version* of `SentiStrength` does not have this shortcoming.

(D_{10}) Sentimental words in interrogative sentences: Typically, negative sentimental words in interrogative sentences (i.e., in questions) either do not express any sentiment or at least weaken the intensity of sentiment [45]. However, we have found instances where `SentiStrength` fails to correctly interpret the sentimental polarities of such interrogative sentences. For example, `SentiStrength` incorrectly identifies negative sentiment in the comment below, although the comment merely includes a question expressing no negative sentiment as indicated by the human raters.

```
"... Did I submit something wrong or duplicate? ..."
```
(Comment ID: 24246)

(D_{11}) Difficulty in dealing with irony and sarcasm: Automatic interpretation of irony in text written in natural language is very challenging, and `SentiStrength` also often fails to detect sentiments from texts, which express irony and sarcasm [45]. For example, due to the presence of the positive sentimental words "Dear God!" in the comment below, `SentiStrength` detects positive sentiment in the sentence, although the comment poster used it in a sarcastic manner and expressed negative sentiment only.

```
"The other precedences are OK; as far as I can tell ... 'zzz'; Dear
God! You mean the intent here is ... gotta confess I just saw the pattern
and jumped to conclusions; hadn't examined the code at all. But you've
just made the job tougher ...?"
```
(Comment ID: 61559)

(D_{12}) Hard to detect subtle expression of sentiments: Text written in natural language can express sentiments without using any inherently sentimental words. The lexical approach of `SentiStrength` fails to identify sentiments in such a text due to its high dependency on the dictionary of lists of words, and not being able to properly capture sentence structure and semantic meanings. Consider the following issue comment, which was labeled with negative sentiment by three human raters although there is no sentimental words in it. Without surprise, `SentiStrength` incorrectly interprets it as a sentimentally neutral text.

```
"Brian; I understand what you say and specification about
'serialization' in XSLT not 'indenting'. As I saied before;
indenting is just the thing that we easily see the structure and data of
XML document. Xalan output is not easy to see that. The last; I think
the example of non-whitespace characters is no
relationship to indenting. non-whitespace characters must not be stripped;
but whitespace characters could be stripped. Regards; Tetsuya Yoshida."
```
(Comment ID: 10134)

4.3 Leveraging Automated Sentiment Analysis

We address the challenges identified from our exploratory study as described in Section 4.2.5.2 and develop a tool particularly crafted for application in the software engineering domain. We call our

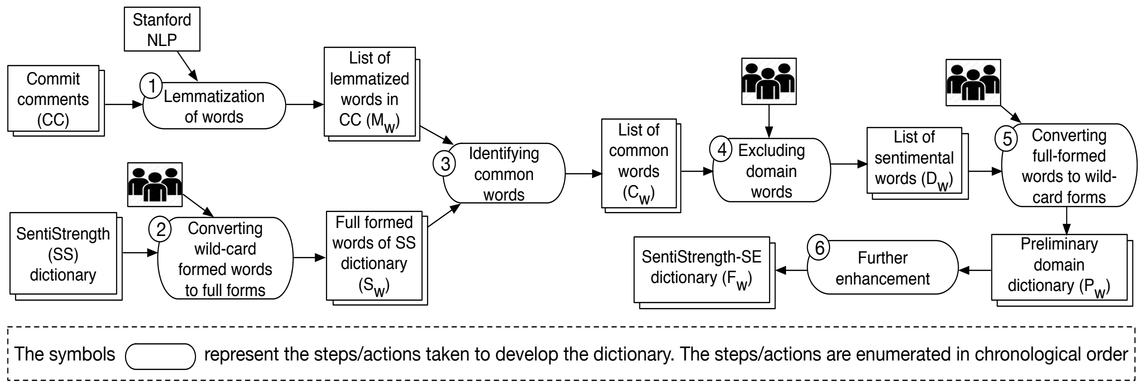

Stanford NLP
Commit comments (CC)
1 Lemmatization of words
List of lemmatized words in CC (M_W)
SentiStrength (SS) dictionary
2 Converting wild-card formed words to full forms
Full formed words of SS dictionary (S_W)
3 Identifying common words
List of common words (C_W)
4 Excluding domain words
List of sentimental words (D_W)
5 Converting full-formed words to wild-card forms
Preliminary domain dictionary (P_W)
6 Further enhancement
SentiStrength-SE dictionary (F_W)
The symbols represent the steps/actions taken to develop the dictionary. The steps/actions are enumerated in chronological order

Figure 4.1: Steps to create the domain dictionary for SentiStrength-SE

tool SentiStrength-SE, which is built on top of the original SentiStrength. We now describe how we mitigate the identified difficulties in developing SentiStrength-SE for improved sentiment analysis in textual artifacts in software engineering.

4.3.1 Creating a New Domain Dictionary for Software Engineering

As reported in Table 4.3, our exploratory study (Section 4.2.5.2) found the domain-specific challenges (difficulties D_1, D_3, D_6) as the most impeding factors against sentiment analysis in software engineering text. Hence, the accuracy of sentiment analysis can be improved by adopting a domain-specific dictionary [143, 144, 145]. We, therefore, first create a domain dictionary for software engineering text to address the issues with domain difficulty.

Figure 4.1 depicts the steps/actions taken to develop the domain dictionary for software engineering text. We collect a large dataset used in the work of Islam and Zibran [38]. This dataset consists of 490 thousand commit messages drawn from 50 open-source projects from GitHub. Using the Stanford NLP tool [146], we extract a set of lemmatized forms of all the words in the commit messages, which is denoted as M_w.

To identify the emotional words from the set M_w, we exploit SentiStrength's existing dictionary. We choose SentiStrength's existing dictionary as the basis for our new one, because, in a recent study [57], SentiStrength's dictionary building method is found superior to other approaches (AFINN [118], MPQA [119], and VADER [120]) for the creation of software engineering domain-specific dictionaries. We identify those words in SentiStrength's original dictionary, which have wild-card forms (i.e., words that have symbol * as suffix) and transform them to their corresponding lemmatized forms using the AFINN [118] dictionary. For example, the entry 'Amaz*' in SentiStrength dictionary is transformed to the words 'Amaze', 'Amazed', 'Amazes' and 'Amazing' as those are found as emotional words in the AFINN dictionary corresponding to that particular entry. The are mainly two reasons for using the AFINN dictionary: (i) the dictionary is very similar to the SentiStrength dictionary as both use the same numeric scale to express sentimental polarities of words, (ii) the

Table 4.4: Inter-rater disagreements in interpretation of sentiments

	Disagreements between Human Raters		
Sentimental Polarity	**A, B**	**B, C**	**C, A**
Positive	11.81%	19.68%	17.32%
Negative	08.62%	10.19%	09.41%
Neutral	18.13%	11.81%	15.69%

`AFINN` dictionary is also widely used in many other studies [147, 148, 149, 57]. If any wild-card form word is not found in AFINN dictionary, we use our own wisdom to convert that word to its lemmatized forms. Thus, by converting all short forms words to full forms and combining those with remaining words in `SentiStrength` dictionary we obtain a set of words S_w. Then, we distinguish a set C_w of words such that $C_w = M_w \cap S_w$. The set C_w ends up containing 716 words, which represent an initial sentimental vocabulary pertinent to the software engineering domain.

We recognize that some of these 716 words are simply software engineering domain-specific technical terms expressing no sentiments in software engineering context, which otherwise would express emotions when interpreted in a non-technical area such as social networking. There also remain other words such as *'Decrease'*, *'Eliminate'* and *'Insufficient'*, which are unlikely to carry sentiments in the software engineering domain. We, therefore, engage three human raters (enumerated as A, B, C) to independently identify these non-sentimental domain words in C_w. Each of these three human raters are computer science graduate students having at least three years of software development experience. A human rater annotates a word as neutral if the word appears to him/her as highly unlikely to express any sentiment when interpreted in the software engineering domain.

In Table 6.6, we present sentiment-wise percentage of cases where the human raters disagree pair-wise. We also measure the degree of inter-rater agreement in terms of *Fleiss-*κ [150] value. The obtained *Fleiss-*κ value 0.739 signifies substantial agreement among the independent raters.

We consider a word as a neutral domain word when two of the three raters identify the word as neutral. Thus, 216 words are identified as neutral domain words, which we exclude from the set C_w resulting in another set D_w of the remaining 500 words. Such neutralization of words for a particular domain is also suggested in several studies [41, 43, 37, 38] in the literature.

Next, we adjust the words in D_w by reverting them to their wild-card forms (if available) to comply with `SentiStrength`'s original dictionary. This new set of words is called as preliminary domain dictionary (P_w), which has 167 positively and 310 negatively polarized words. This preliminary dictionary is further enhanced according to the description below to create the `SentiStrength-SE` dictionary (F_w).

4.3.1.1 Further Enhancements to the Preliminary Domain Dictionary

We further extend the newly developed preliminary dictionary in the light of our observations during the exploratory study described in Section 4.2.

Extension with new sentimental words and negations: During our exploratory study, we find several informal sentimental words such as, 'Woops', 'Uhh', 'Oops' and 'zzz', which are not included in the original dictionary. The formal word 'Apology' is also missing from the dictionary. We have added to the dictionary of our `SentiStrength-SE` all these missing words as sentimental terms with appropriate sentimental polarities, which mitigate the difficulty D_6.

In addition, we also add to the dictionary the missing shortened from of negation words as mentioned in the discussion of difficulty D_5 in Section 4.2.5.2.

Discarding the letter 'X' from dictionary: We exclude the letter 'X' from our domain dictionary of `SentiStrength-SE` to save lexical sentiment analysis from the difficulty D_3 as described in Section 4.2.5.2.

4.3.2 Inclusion of Heuristics in Computation of Sentiments

While the creation of the new domain dictionary is a vital step towards automated sentiment analysis in software engineering text, we realize that the computations for sentiment detection also need improvements. Thus, in the implementation of our domain-specific `SentiStrength-SE`, we incorporate a number of heuristics in the computation, which we describe below.

4.3.2.1 Addition of Contextual Sense to Minimize Ambiguity

Recall that, in the creation of our initial domain dictionary, we neutralized 216 words on the basis of the judgements from three independent human raters. However, the neutralization of words is not always appropriate. For example, in the software engineering domain, the word 'Fault' typically indicates a program error and expresses neutral sentiment. However, the same word can also convey negative sentiment as found in the following comment.

> `"As WING ... My fault: I cannot reproduce after holidays ... I might add that one; too" the word 'Fault' expresses negative sentiment of the comment poster."` (Comment ID: 4694)

Again, the word 'Like' expresses positive sentiment if it is used as *"I like"*, *"We like"*, *"He likes"*, and *"They like"*. In the most other cases the word 'Like' is used as *preposition* or *subordinating conjunction* and the word can safely be considered sentimentally neutral. For example, the following comment used the word 'Like' without expressing any sentiment.

> `"Looks like a user issue to me ..."` (Comment ID: 40844)

We can observe from the above examples that some of the 216 neutralized words can actually express sentiments when those are preceded by *pronouns* referring to a person or a group of persons, e.g., 'I', 'We', 'My', 'He', 'She', 'You' and possessive pronouns such as 'My' and 'Your'. This contextual information is taken into account in `SentiStrength-SE` to appropriately deal with the contextual use of those words in software engineering field to minimize the difficulties D_1 and

D_2. The complete list of such words is given in the SentiStrength-SE dictionary file named 'ModifiedTermsLookupTable', which are also vetted by the three raters. Note that to determine the Part-Of-Speech (POS) of words in sentences, we apply the Stanford POS tagger [146].

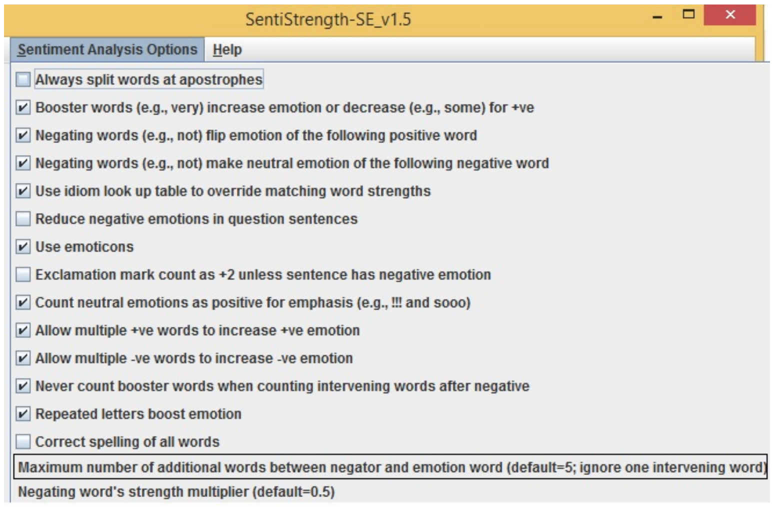
4.3.2.2 Bringing Neutralizers in Effect

Our observations from the exploratory study (as presented in Section 4.2) reveal that sentiment of a word can be neutralized if that word is preceded by any of the neutralizer words such as, 'Would', 'Could', 'Should', and 'Might'. For example in the sentence "*It would be good if the test could be completed soon*" the positive sentimental word 'Good' does not express any sentiment as neutralized by the preceding word 'Would'. We add a method in SentiStrength-SE to enable it correctly detect uses of such neutralizer words in sentences to be more accurate in sentiments detection. This helps in minimizing the difficulty D_2 described before.

4.3.2.3 Integration of a Preprocessing Phase

To minimize the difficulties D_4, D_7, D_8, and D_9 (as described in Section 4.2.5.2), we include a preprocessing phase to SentiStrength-SE as its integral part. Before computation for any given input text, SentiStrength-SE applies this preprocessing phase to filter out numeric characters and certain copy-pasted contents such as code snippets, URLs and stack traces. To locate code snippets, URL's and stack traces in text, we use regular expressions similar to the approach proposed by Bettenburg et al. [151]. In addition, a spellchecker [152] is also included to deal with the difficulty D_7 in identifying and rectifying misspelled English words. Spell checking also complements our regular expression based method in approximate identification of identifier names in code snippets.

To mitigate the difficulty D_9 in particular, the preprocessing phase also converts all the letters of a comment to small letters. However, converting all the letters to small letters can also cause failure of the detection of the proper nouns such as the names of developers and systems, which is also important as discussed in the description of difficulty D_9 in Section 4.2.5.2. From our exploratory study, we have observed that the developers typically mention their colleagues' names in comments immediately after some sort of salutation words such as 'Dear', 'Hi', 'Hello', 'Hellow' or after the character '@'. Hence, in addition to converting all letters to lower case, the preprocessing phase also discards those words, which are placed immediately after any of those salutation words or the character '@'. In addition, SentiStrength-SE maintains the flexibility to allow the user to instruct the tool to consider any particular words as neutral in sentiment, in case an inherently sentimental word must be recognized as proper noun, for example, to deal with the situation where a sentimental word is used as a system's name.



- ☐ Always split words at apostrophes
- ☑ Booster words (e.g., very) increase emotion or decrease (e.g., some) for +ve
- ☑ Negating words (e.g., not) flip emotion of the following positive word
- ☑ Negating words (e.g., not) make neutral emotion of the following negative word
- ☑ Use idiom look up table to override matching word strengths
- ☐ Reduce negative emotions in question sentences
- ☑ Use emoticons
- ☐ Exclamation mark count as +2 unless sentence has negative emotion
- ☑ Count neutral emotions as positive for emphasis (e.g., !!! and sooo)
- ☑ Allow multiple +ve words to increase +ve emotion
- ☑ Allow multiple -ve words to increase -ve emotion
- ☑ Never count booster words when counting intervening words after negative
- ☑ Repeated letters boost emotion
- ☐ Correct spelling of all words

Maximum number of additional words between negator and emotion word (default=5; ignore one intervening word)

Negating word's strength multiplier (default=0.5)

Figure 4.2: Default configuration of parameters in our `SentiStrength-SE`

4.3.2.4 Parameter Configuration for Better Handling of Negations

We carefully set a number of configuration parameters as defaults to our `SentiStrength-SE` tool as shown in Figure 4.2. This default configuration of `SentiStrength-SE` is different from that of the original `SentiStrength`. Particularly, to mitigate the difficulty D_5 in dealing with negations, the negation's configuration parameter marked with a black rectangle in Figure 4.2 is set to five in `SentiStrength-SE`, which enables the tool detecting negations over a larger range of proximity allowing zero to five intervening words between a negation and a sentimental word, as was also suggested in a previous study [144].

4.4 Empirical Evaluation of `SentiStrength-SE`

While making the design and tuning decisions to develop `SentiStrength-SE`, we remain careful about the possibility that the application of a particular heuristic for improvement in one area might have side-effects causing performance degradation in another criteria. We empirically evaluate our domain-specific techniques and the accuracy of domain-specific `SentiStrength-SE` in several phases around seven research questions.

Dataset: For empirical evaluation of our `SentiStrength-SE`, we use the 5,600 issue comments in Group-2 and Group-3 of the "Gold Standard" dataset introduced in Section 4.2.1. The *ground-truth* about the sentimental polarities of those issue comments are determined based on the manual evaluations by human raters as described in Section 4.2.3. Before conducting evalution, we present textual characteristics of Group-2 and Group-3 datasets in Table 4.4, which indicates no substantial differences in the characteristics of the two datasets.

Table 4.5: Textual characteristics of the words in the datasets

Datasets	Number of Distinct Words	Complexity Factor (Lexical Density)	Number of Sentences	Average Sentence Length (in words)	Number of Sentences per Comment
Group-2	5,295	21.00%	5,671	8.23	3.54
Group-3	5,527	24.30%	4,000	8.26	1.00

Metrics: The accuracy of sentiment analysis is measured in terms of *precision*, *recall*, and *F-score* computed for each of the three sentimental polarities (i.e., positivity, negativity and neutrality). Given a set S of textual contents, *precision* ($\wp$), *recall* ($\Re$), and *F-score* ($\Finv$) for a particular sentimental polarity e is calculated as follows:

$$\wp = \frac{\mid \mathcal{S}_e \cap \mathcal{S}'_e \mid}{\mid \mathcal{S}'_e \mid}, \qquad \Re = \frac{\mid \mathcal{S}_e \cap \mathcal{S}'_e \mid}{\mid \mathcal{S}_e \mid}, \qquad \Finv = \frac{2 \times \wp \times \Re}{\wp + \Re}$$

where $\mathcal{S}_e$ represents the set of texts having sentimental polarity e, and $\mathcal{S}'_e$ denotes the set of texts that are detected (by tool) to have the sentimental polarity e.

Statistical Test of Significance: We apply non-parametric *McNemar's* test [153] to verify the statistical significance in the difference of the results obtained by two tools, say $\mathcal{T}_a$ and $\mathcal{T}_b$. As the non-parametric test does not require normal distribution of data, this test suits well for our purpose. We perform a *McNemar's* test on 2×2 contingency table derived from the results obtained from tools $\mathcal{T}_a$ and $\mathcal{T}_b$. The structure of such a contingency table is shown in Table 4.6.

Table 4.6: Structure of 2×2 contingency matrix of McNemar's test for tools $\mathcal{T}_a$ and $\mathcal{T}_b$

# of comments misclassified by both $\mathcal{T}_a$ and $\mathcal{T}_b$	n_{00}	n_{01}	# of comments misclassified by $\mathcal{T}_b$ but not by $\mathcal{T}_a$
# of comments misclassified by $\mathcal{T}_a$ but not by $\mathcal{T}_b$	n_{10}	n_{11}	# of comments correctly classified by both $\mathcal{T}_a$ and $\mathcal{T}_b$

Let, F_a and F_b denote the sets of misclassified comments by $\mathcal{T}_a$ and $\mathcal{T}_b$ respectively. In the contingency table (Table 4.6), n_{00} represents the number of issue comments misclassified by both $\mathcal{T}_a$ and $\mathcal{T}_b$ (i.e., $n_{00} = |F_a \cap F_b|$), n_{01} represents the number of comments misclassified by $\mathcal{T}_b$ but not by $\mathcal{T}_a$ (i.e., $n_{01} = |F_b - F_a|$), n_{10} represents the number of comments misclassified by $\mathcal{T}_a$ but not by $\mathcal{T}_b$ (i.e., $n_{01} = |F_a - F_b|$), and n_{11} represents the number of comments correctly classified by both the tools. Let, S denote the set of all the issue comments correctly classified according to the ground-truth. Thus, $n_{11} = S - (F_a \cup F_b)$. The superiority of tool $\mathcal{T}_b$ over the tool $\mathcal{T}_a$ is observed, if $n_{10} > n_{01}$. Otherwise, $\mathcal{T}_a$ is considered superior if $n_{01} > n_{10}$. Such observed superiority is considered statistically significant, if the *p-value* obtained from a *McNemar's* test is less than a pre-specified significance level α. In our work, we set $\alpha = 0.001$, which a reasonable setup widely used in the literature.

4.4.1 Head-to-head Comparison Using a Benchmark Dataset

We compare our software engineering domain-specific `SentiStrength-SE` with the original `SentiStrength` [45] tool and two other toolkits `NLTK` [46] and `Stanford NLP` [47]. To the best of our knowledge, these are the only *domain independent* tools/toolkits attempted in the past for sentiment analysis in software engineering text [41, 40, 50, 154]. In particular, we address the following research question:

RQ1: *Does our domain-specific `SentiStrength-SE` outperform the existing domain independent tools for sentiment analysis in software engineering text?*

We write a Python script to import `NLTK` sentiment analysis package [155, 46] and run it on texts to determine the sentimental polarities of those. `NLTK` determines the probability of positivity, negativity and neutrality of a text. In addition, its also provides a compound value C_v, which ranges between -1 to +1. A text contains positive sentiments if $C_v > 0$, a text will have negative sentiments if $C_v < 0$. Otherwise, a text is considered sentimentally neutral when $C_v = 0$. Similar procedure is also followed in another study [155] to determine sentiments of texts using `NLTK`.

We develop a Java program using the JAR of the `Stanford NLP` tool to apply it on texts to determine their sentimental polarities. For a text `Stanford NLP` provides a sentiment score S_v between zero to four where $0 \leq S_v \leq 1$ indicates negative sentiment, $3 \leq S_v \leq 4$ indicates positive sentiment and $S_v = 2$ indicates neutral sentiment of the text [156].

We separately operate each of the selected tools and our `SentiStrength-SE` on the Group-2 and Group-3 portions of the “Gold Standard" dataset. Recall that Group-2 and Group-3 datasets contain 1,600 and 4,000 issue comments respectively. For each of the three sentimental polarities (i.e., positivity, negativity, and neutrality), we compare the tools' outcome with the ground-truth and separately compute precision, recall, and F-score for all the tools with respect to each dataset. Table 4.7 presents the precision ($\wp$), recall ($\Re$), and F-score ($\Finv$) of all the tools in the detection of positive, negative and neutral sentiments, and also the average over all these three sentimental polarities. The highest metric values are highlighted in bold.

Notice that for the Group-2 dataset, our `SentiStrength-SE` consistently achieves the highest precision, recall and F-scored compared to the rest other tools.

For the Group-3 dataset, `SentiStrength-SE` achieves the highest precision and F-score in detecting negative sentiments and it achieves the highest recall and F-score in the detection of neutral sentiments. In those few cases, where `SentiStrength-SE` does not achieve the best results, it remains at the second best or marginally close to the best. In the detection of positive

Table 4.7: Head-to-head comparison of performances of the four tools/toolkits

Data	Senti-ments	Met.	Senti-Strength-SE	Senti-Strength	NLTK	Stanford NLP
Group-2	Positive	$\wp$	**88.86%**	74.48%	69.47%	79.77%
		$\Re$	**98.81%**	**98.81%**	81.55%	71.67%
		Ⅎ	**93.57%**	84.93%	75.0%	75.50%
	Negative	$\wp$	**53.42%**	28.22%	40.46%	13.28%
		$\Re$	**97.66%**	**97.66%**	54.69%	88.28%
		Ⅎ	**69.06%**	43.78%	46.51%	23.08%
	Neutral	$\wp$	**98.14%**	96.83%	69.53%	63.70%
		$\Re$	**83.00%**	52.42%	50.86%	25.57%
		Ⅎ	**89.94%**	68.01%	58.75%	36.49%
Group-3	Positive	$\wp$	41.80%	31.69%	20.32%	**69.47%**
		$\Re$	82.04%	**87.79%**	86.33%	81.55%
		Ⅎ	55.39%	46.58%	32.89%	**75.03%**
	Negative	$\wp$	**68.61%**	47.61%	50.65%	40.46%
		$\Re$	71.00%	**78.40%**	70.24%	54.69%
		Ⅎ	**69.78%**	59.25%	58.86%	46.51%
	Neutral	$\wp$	90.64%	**91.28%**	91.17%	69.53%
		$\Re$	**80.05%**	56.16%	45.78%	50.86%
		Ⅎ	**85.02%**	69.54%	60.96%	58.74%
Overall average accuracy		$\wp$	**73.58%**	61.69%	56.93%	56.04%
		$\Re$	**85.43%**	78.54%	64.91%	62.10%
		Ⅎ	**79.06%**	62.02%	55.50%	52.56%

sentiments in Group-3 dataset, `Satnford NLP` achieves the highest precision and recall, where our `SentiStrength-SE` yields the second best results. Similarly, the original `SentiStrength` achives the highest recall in the detection of negative sentiments in Group-3 dataset, and again our `SentiStrength-SE` obtains the second best result. The highest precision (91.28% achieved by the original `SentiStrength`) for neutral sentiments is only 0.64% higher than that of our `SentiStrength-SE`.

Thus, if we consider the overall average accuracy, as presented at the bottom of the table, it becomes evident that our `SentiStrength-SE` performs the best, followed by the original `SentiStrength` and `NLTK`. Notice that the overall precision, recall and F-score of our `SentiStrength-SE` are substantially higher than those of the second-best performing tool (i.e., the original `SentiStrength`).

Table 4.8: Contingency matrix of McNemar's test in comparison between the original `SentiStrength` and `SentiStrength-SE`

# of comments misclassified by both $\mathcal{T}_a$ and $\mathcal{T}_b$	748	365	# of comments misclassified by $\mathcal{T}_b$ but not by $\mathcal{T}_a$
# of comments misclassified by $\mathcal{T}_a$ but not by $\mathcal{T}_b$	**1,527**	2,924	# of comments correctly classified by both $\mathcal{T}_a$ and $\mathcal{T}_b$

Here, $\mathcal{T}_a$ = original `SentiStrength` and $\mathcal{T}_b$ = `SentiStrength-SE`

To verify whether the observed performance difference between `SentiStrength-SE` and the original `SentiStrength` is statistically significant, we perform a *McNemar's* test between the results of these two tools. For both datasets Group-2 and Group-3, we compute n_{00}, n_{01}, n_{10} and n_{11} according to their specifications described in Table 4.6. In Table 7.8, we present the contingency matrix computed for the *McNemar's* test. We observe superiority of $\mathcal{T}_b$ (`SentiStrength-SE`) in the contingency table as $n_{10} > n_{01}$. According to the p-value ($p = 2.2 \times 10^{-16}, p < \alpha$) obtained from the test, the observed difference in the superior performance of `SentiStrength-SE` is statistically significant. Based on these observations and statistical test, we now derive the answer to the research question RQ1 as follows:

Ans. to RQ1: *In the detection of sentiments in software engineering text, our domain-specific `SentiStrength-SE` significantly outperforms the domain independent `NLTK`, `Stanford NLP`, and the original `SentiStrength`.*

4.4.2 Comparison with respect to Human Raters' Disagreements

Recall that the issue comments in the "Gold Standard" dataset are annotated with sentiments as identified by independent human raters. There are disagreements among human raters in the identification of sentiments in some issue comments. While humans disagree about sentiments in some issue comments, it is likely that the automated tools will also produce different outcomes resulting in varied precision and recall.

We, therefore, investigate to what extend the agreements and disagreements of annotations among human raters cause deviation of results of the head-to-head comparison of tools as described in the previous section. Particularly, we want to verify whether our domain-specific `SentiStrength-SE` still outperforms the other domain-independent tools when the rater' agreements and disagreements are taken into account. Here, we address the following research question:

RQ2: *Does the accuracy of `SentiStrength-SE` largely vary compared to its domain independent counterparts when the agreements and disagreements among human raters are taken into account?*

For this investigation, we use the Group-2 dataset where each issue comment was independently annotated by three human raters. We distinguish two sets of issue comments from this Group-2 dataset.

i) *Set-A*: containing those issue comments for which all the three human raters agreed on the sentiments expressed in those comments. This set contains 1,210 issue comments.

ii) *Set-B*: consisting of those issue comments for which two of the three raters agreed on the sentiments expressed in those comments. This set contains 357 issue comments.

We formulate the following null and alternative hypotheses to determine the statistical significance of improved performances of the best tool.

Null Hypothesis-1 (H_o^1): There is no significant difference in the performance of `SentiStrength-SE` compared to the other tools in sentiment detection in the issue comments in *Set-A*.

Alternative Hypothesis-1 (H_a^1): There exist significant differences in the performance of `SentiStrength-SE` compared to the other tools in sentiment detection in the issue comments in *Set-A*.

Null Hypothesis-2 (H_o^2): There is no significant difference in the performance of `SentiStrength-SE` compared to the other tools in sentiment detection in the issue comments in *Set-B*.

Alternative Hypothesis-2 (H_a^2): There exist significant differences in the performance of `SentiStrength-SE` compared to the other tools in sentiment detection in the issue comments in *Set-B*.

We now examine whether these hypotheses hold true with respect to the four tools we compare. In the similar way as of the head-to-head comparison described in the previous section, we separately run all the four tools including our `SentiStrength-SE` on *Set-A* and *Set-B* issue comments. For each of the three sentimental polarities (i.e., positivity, negativity, and neutrality), we compute precision ($\wp$), recall ($\Re$), and F-score ($\Finv$) for each of the tools separately over the issue comments in both *Set-A* and *Set-B*.

For the issue comments in both *Set-A* and *Set-B*, the best tool *must* exhibit the significantly improved performances compared to other tools. For testing our hypotheses, in each of *Set-A* and *Set-B* datasets, we first identify the two tools producing better results among all the four tools. Then, we examine if there is any significant difference in the performances of the best tool and the second best one. If there exist significant differences *between* the accuracies of the top performing two tools, discovering such would suffice for demonstrating the existence of significant difference of the best performing tool against the other tools.

Table 4.9 presents the metrics' values of all the tools in the detection of positive, negative and neutral sentiments for each set of the issue comments. As seen in Table 4.9, for the issue comments in *Set-A*, `SentiStrength-SE` consistently achieves the highest precision, recall and F-score in the detection of positive, negative and neutral sentiments. The overall average accuracies indicate that the original `SentiStrength` achieves the second best accuracies in the *Set-A* dataset.

Now, we perform a *McNemar's* test between the accuracies of `SentiStrength-SE` and the original `SentiStrength` for the issue comments in *Set-A* dataset. The contingency table for the test is presented in Table 4.10. According to the contingency table, `SentiStrength-SE` ($\mathcal{T}_b$) performs better as $n_{10} > n_{01}$. The performance difference is found to be statistically significant with $p = 2.2 \times 10^{-16}$ and $p < \alpha$. Thus, the *McNemar's* test rejects our first null hypothesis (H_0^1). Therefore, the first alternative hypothesis (H_a^1) holds true.

Table 4.9: Comparison of tools' accuracies for *Set-A* and *Set-B* issue comments

Data	**Senti-ments**	**Met.**	**Senti-Strength-SE**	**Senti-Strength**	**NLTK**	**Stanford NLP**
Set-A	Positive	$\wp$	**90.15%**	70.46%	67.9%	83.39%
		$\Re$	**95.33%**	91.20%	61.58%	76.66%
		$\Finv$	**92.67%**	79.50%	64.17%	79.89%
	Negative	$\wp$	**53.66%**	28.17%	49.45%	9.66%
		$\Re$	**100.00%**	**100.00%**	54.55%	86.36%
		$\Finv$	**69.84%**	43.96%	51.87%	17.38%
	Neutral	$\wp$	**99.44%**	99.43%	77.00%	67.61%
		$\Re$	**91.15%**	58.84%	54.08%	28.40%
		$\Finv$	**95.12%**	73.93%	63.53%	40.00%
	Overall average accuracy	$\wp$	**81.08%**	66.02%	64.78%	53.55%
		$\Re$	**95.49%**	83.35%	56.74%	63.81%
		$\Finv$	**85.88%**	65.79%	59.86%	45.76%
Set-B	Positive	$\wp$	**72.48%**	63.64%	67.32%	64.91%
		$\Re$	96.89%	**97.92%**	70.46%	57.13%
		$\Finv$	**82.92%**	77.14%	68.86%	60.98%
	Negative	$\wp$	**51.75%**	21.63%	50.74%	20.83%
		$\Re$	**96.72%**	60.65%	55.73%	90.16%
		$\Finv$	**67.42%**	31.89%	53.12%	33.85%
	Neutral	$\wp$	83.92%	**86.21%**	38.24%	38.88%
		$\Re$	**40.87%**	21.74%	33.91%	12.17%
		$\Finv$	**54.97%**	34.72%	35.94%	18.54%
	Overall average accuracy	$\wp$	**69.38%**	57.16%	52.10%	41.54%
		$\Re$	**78.16%**	60.10%	53.37%	53.15%
		$\Finv$	**68.44%**	58.59%	52.72%	37.79%

Table 4.10: Contingency matrix of McNemar's test between the accuracies of `SentiStrength-SE` and the original `SentiStrength` in *Set-A* dataset

# of comments misclassified by both $\mathcal{T}_a$ and $\mathcal{T}_b$	84	22	# of comments misclassified by $\mathcal{T}_b$ but not by $\mathcal{T}_a$
# of comments misclassified by $\mathcal{T}_a$ but not by $\mathcal{T}_b$	**511**	593	# of comments correctly classified by both $\mathcal{T}_a$ and $\mathcal{T}_b$

Here, $\mathcal{T}_a$ = original `SentiStrength` and $\mathcal{T}_b$ = `SentiStrength-SE`

Again, as seen in Table 4.9, for the issue comments in *Set-B*, `SentiStrength-SE` achieves the highest F-score in detecting sentiments of all the three polarities. The precision and recall of `SentiStrength-SE` is also the highest in all cases except for only two. The recall of `SentiStrength-SE` for positive sentiments is 96.89%, which is the second best and only 1.03% lower than the highest. Similarly, the precision of our `SentiStrength-SE` in detecting neutral sentiments is 83.92%, which is also next to the best and only 2.29% lower than the best. Also, with respect to the overall average accuracies, `SentiStrength` can be considered to have achieved the second best performance for the *Set-B* dataset.

Table 4.11: Contingency matrix of McNemar's test between the accuracies of `SentiStrength-SE` and the original `SentiStrength` in *Set-B* dataset

# of comments misclassified by both $\mathcal{T}_a$ and $\mathcal{T}_b$	97	20	# of comments misclassified by $\mathcal{T}_b$ but not by $\mathcal{T}_a$
# of comments misclassified by $\mathcal{T}_a$ but not by $\mathcal{T}_b$	**135**	105	# of comments correctly classified by both $\mathcal{T}_a$ and $\mathcal{T}_b$

Here, $\mathcal{T}_a$ = original `SentiStrength` and $\mathcal{T}_b$ = `SentiStrength-SE`

Similar to the *Set-A* dataset, for the *Set-B*, we carry out a *McNemar's* test between the accuracies of `SentiStrength-SE` and the original `SentiStrength` to determine whether or not there is any statistical significant differences between the performances of these two tools. The contingency matrix for the test is presented in Table 4.11. According to the contingency matrix, `SentiStrength-SE` ($\mathcal{T}_b$) outperforms the original `SentiStrength` ($\mathcal{T}_a$) with $n_{10} > n_{01}$. The *McNemar's* test with the contingency matrix of Table 4.11 obtains $p = 2.2 \times 10^{-16}$ and thus $p < \alpha$. Thus, our second null hypothesis (H_0^2) is rejected and the second alternative hypothesis (H_a^2) holds true, which indicates that the superior performance of `SentiStrength-SE` over the original `SentiStrength` is statistically significant for the issue comments in the *Set-B* dataset.

Thus, for both the *Set-A* and *Set-B* datasets, `SentiStrength-SE` significantly outperforms the next best performer `SentiStrength`. Based on our observations and results from the statistical tests over both the *Set-A* and *Set-B* datasets, we now derive the answer to the research question RQ2 as follows:

Ans. to RQ2: *When the agreements and disagreements among human raters are taken into account, our domain-specific `SentiStrength-SE` still maintains significantly superior (compared to its domain independent counterparts) accuracies in detecting sentiments in software engineering text.*

4.4.3 Evaluating the Contribution of Domain Dictionary

The newly developed software engineering domain dictionary is a major component of `SentiStrength-SE`. Here, we carry out a quantitative evaluation to verify the contribution of the domain dictionary in detecting sentiments in software engineering texts accurately. Especially, we address the following research question:

RQ3: *Does the domain-specific dictionary in `SentiStrength-SE` really contribute to improved sentiment analysis in software engineering text?*

For this particular evaluation, we again use the Group-2 and Group-3 datasets introduced before. We invoke the original `SentiStrength` for detecting sentiments in issue comments in these datasets. Then, we operate `SentiStrength` making it use our newly developed domain dictionary and invoke it for sentiment detection in the same issue comments. We use `SentiStrength*` to refer to the variant of the original `SentiStrength` that is forced to use our domain dictionary instead of its original one. For each of the three sentimental polarities, we separately compute and compare the precision, recall, and F-score resulting from the tools in each dataset.

Table 4.12: Comparison of performances between **SentiStrength** and **SentiStrength***

Data	**Sentiment**	**Met.**	**SentiStrength**	**SentiStrength***
Group-2	Positive	$\wp$	74.48%	87.56%
		$\Re$	98.81%	98.28%
		$\Finv$	84.93%	**92.61%**
	Negative	$\wp$	28.22%	**53.19%**
		$\Re$	97.66%	97.65%
		$\Finv$	43.78%	**68.87%**
	Neutral	$\wp$	96.83%	**97.94%**
		$\Re$	52.42%	**81.85%**
		$\Finv$	68.01%	**89.18%**
Group-3	Positive	$\wp$	31.69%	**40.44%**
		$\Re$	**87.79%**	82.01%
		$\Finv$	46.58%	**54.16%**
	Negative	$\wp$	47.61%	**69.10%**
		$\Re$	**78.40%**	72.65%
		$\Finv$	59.25%	**70.83%**
	Neutral	$\wp$	91.28%	91.22%
		$\Re$	56.16%	**79.54%**
		$\Finv$	69.54%	**84.98%**
Overall average accuracy		$\wp$	61.69%	**73.24%**
		$\Re$	78.54%	**85.33%**
		$\Finv$	62.02%	**76.77%**

Here, `SentiStrength*` is forced to use our domain dictionary instead of its own one.

4.4.3.1 Comparison between the `SentiStrength` and `SentiStrength*`

If our domain dictionary actually contributes to improved sentiment analysis in software engineering text, `SentiStrength*` must perform better than the original `SentiStrength`. In Table 4.4.2, we present the precision ($\wp$), recall ($\Re$), and F-score ($\Finv$) obtained in detection of each sentimental polarity. In the table, substantial (i.e., more that 1%) differences are marked in bold.

As seen in Table 4.4.2, in every case, `SentiStrength`* achieves higher F-score than the original `SentiStrength`. Moreover, `SentiStrength`* shows *much higher precision* in all cases except for neutral comments in Group-3 dataset. For the neutral comments in Group-3 dataset, the precision of the original `SentiStrength` is marginally higher by only 0.06%. In all the cases across datasets the precision, recall, and F-score of `SentiStrength`* is higher or comparable to those of the original `SentiStrength`. Only in two of the 18 cases (i.e., the recall for positive and negative sentiments in Group-3 dataset), the original `SentiStrength`'s performance is perceived (substantially) better than `SentiStrength`*. These observations are also reflected in the *overall average accuracies* presented in the bottom three rows of the table. The *overall average accuracies* indicate superior performance of `SentiStrength`* over the original `SentiStrength`. Hence, the observed accuracy of `SentiStrength`* is substantially higher when it is forced to use our new domain dictionary instead of its original one. To determine the statistical significance of our observations, we perform another *McNemar's* test between the results of `SentiStrength`* and the original `SentiStrength`. As such, we formulate our null and alternative hypotheses as follows:

Null Hypothesis-3 (H_0^3): There is no *significant* difference between the accuracies of the original `SentiStrength` and `SentiStrength`*.

Alternative Hypothesis-3 (H_a^3): There exist *significant* differences between the accuracies of the original `SentiStrength` and `SentiStrength`*.

Table 4.13: *McNemar's* test between `SentiStrength` and `SentiStrength`*

# of comments misclassified by both $\mathcal{T}_a$ and $\mathcal{T}_b$	748	334	# of comments misclassified by $\mathcal{T}_b$ but not by $\mathcal{T}_a$
# of comments misclassified by $\mathcal{T}_a$ but not by $\mathcal{T}_b$	**1,527**	2,955	# of comments correctly classified by both $\mathcal{T}_a$ and $\mathcal{T}_b$

Here, $\mathcal{T}_a$ = original `SentiStrength` and $\mathcal{T}_b$ = `SentiStrength*`

The contingency matrix for the *McNemar's* test is presented in Table 4.4.3.1. As seen in the contingency matrix, `SentiStrength*` ($\mathcal{T}_b$) exhibits higher accuracies (compared to the original `SentiStrength`) as $n_{10} > n_{01}$. The test obtains $p = 2.2 \times 10^{-16}$ where $p < \alpha$. Thus, the test rejects our null hypothesis (H_0^3). Hence the alternative hypothesis (H_a^3) holds true indicating that the difference is statistically significant. Based on these observations and the statistical test, we conclude that our newly created domain dictionary indeed contributes to statistically significant improvements in sentiment analysis. Hence, we answer the research question RQ3 as follows:

Ans. to RQ3: *Our newly created domain dictionary makes statistically significant contributions to the improvement of sentiment analysis in software engineering domain.*

4.4.4 Our Domain Dictionary vs. `SentiStrength`'s Optimized Dictionary

The original `SentiStrength` has a feature that facilitates optimizing its dictionary for a particular domain [157]. We want to verify how our domain dictionary perform in comparison with `SentiStrength`'s dictionary optimized for software engineering text. In particular, we address the following research question:

RQ4: *Can `SentiStrength`'s dictionary optimized for software engineering text perform better than `SentiStrength-SE`'s domain-specific dictionary we created?*

4.4.4.1 Optimizing `SentiStrength`'s Dictionary

`SentiStrength`'s original dictionary can be optimized for a particular domain by feeding it with a set of annotated pieces of texts. To optimize the `SentiStrength`'s dictionary for software engineering domain, we use a dataset consists of Stack Overflow posts/comments related to software engineering. This Stack Overflow posts (SOP) dataset contains total 4,423 comments [55, 130]. Each comment in the SOP dataset is assigned appropriate sentimental polarities (i.e., *positive*, *negative*, *neutral*) depending on which ones it expresses. Thus, 35% of posts are labeled with positive sentiment and 27% are labeled with negative sentiment while 38% of the posts are labeled as neutral in sentiment [55].

Simple annotations with sentimental polarity labels is not enough `SentiStrength` to be able to use the dataset for optimizing its dictionary. For this purpose, `SentiStrength` requires a pair of integer sentiment scores $\langle \varrho_c, \mu_c \rangle$ pre-assigned to each comment C where $+1 \leq \varrho_c \leq +5$ and $-5 \leq \mu_c \leq -1$. The interpretation of these score is similar to what is described in Section 4.2.4. ϱ_c and μ_c respectively represent the positive and negative sentimental scores pre-assigned to the given text C. A given text C labeled to have have positive sentiment, must be assigned a positive sentimental score $\varrho_c > +1$. A higher ϱ_c indicates a higher intensity/strength of the positive emotion. Similarly, a text labeled to have negative sentiment must be assigned a negative sentimental score $\mu_c < -1$. A lower μ_c signifies a higher intensity/strength of the negative emotion expressed in text C. A text labeled as neutral in sentiment, must be assigned sentimental scores $\langle 1, -1 \rangle$.

Table 4.14: Examples of assigning sentiment scores to labeled comments

Comment Text	**Sentiments Labeled by Human Raters**	**Sentimental Scores** $\langle \varrho_c, \mu_c \rangle$
@DrabJay: excellent suggestion! Code changed. :-)	Positive	$\langle +3, -1 \rangle$
That really stinks! I was afraid of that...	Negative	$\langle +1, -3 \rangle$
A few but they all seem proprietary	Neutral	$\langle +1, -1 \rangle$

According to the requirements described above, we derive the sentimental scores for each of the comments in the SOP dataset. For a comment C having positive sentiment, we set $\varrho = +3$. Similarly, we set $\mu = -3$ for a comment expressing negative sentiment. Instead of using extreme values from the domains of ϱ and μ, we choose the ones at the medians. In Table 4.4.4.1, we present examples demonstrating how we assign sentiment scores to the labeled comments in the SOP dataset. This dataset is then fed to the original `SentiStrength` for optimizing its dictionary for software engineering text. Thus, we produce another variant of the original `SentiStrength`. We refer to this variant with the optimized dictionary as $\texttt{SentiStrength}^{O}$.

4.4.4.2 Comparison between $\texttt{SentiStrength}^{O}$ and `SentiStrength*`

$\texttt{SentiStrength}^{O}$ and `SentiStrength*` only differ in their dictionaries. $\texttt{SentiStrength}^{O}$ uses the optimized dictionary while `SentiStrength*` uses the dictionary we created for `SentiStrength-SE`. Thus, comparing between $\texttt{SentiStrength}^{O}$ and `SentiStrength*` implies a comparison between `SentiStrength`'s optimized dictionary and `SentiStrength-SE`'s software engineering domain dictionary we created.

We invoke $\texttt{SentiStrength}^{O}$ for detecting sentiments in issue comments in Group-2 and Group-3 datasets and compute the values of precision ($\wp$), recall ($\Re$), and F-score ($\daleth$) in detection of each sentimental polarity. We present the computed metrics values for $\texttt{SentiStrength}^{O}$ and `SentiStrength*` side by side in Table 4.4.4.2.

In Table 4.4.4.2, we see that `SentiStrength*` always achieves higher F-score than $\texttt{SentiStrength}^{O}$. Moreover, `SentiStrength*` achieves *much higher precision* in all cases except for neutral comments in Group-3 dataset. For the neutral comments in Group-3 dataset, the precision of `SentiStrength*` lower than that of $\texttt{SentiStrength}^{O}$ marginally by only 0.49%. The recall of `SentiStrength*` is also substantially higher than or nearly equal to that of $\texttt{SentiStrength}^{O}$ in 16 of 18 cases. Finally, the overall average accuracies, as presented at the bottom three rows of the table, indicate that the overall precision, recall, and F-score of `SentiStrength*` are substantially higher than $\texttt{SentiStrength}^{O}$.

To determine the statistical significance of our observations, we perform another *McNemar's* test between the results of $\texttt{SentiStrength}^{O}$ and `SentiStrength*`. Thus, we formulate our null and alternative hypotheses as follows:

Null Hypothesis-4 (H_0^4): There is no significant difference between the accuracies of $\texttt{SentiStrength}^{O}$ and `SentiStrength*`.

Alternative Hypothesis-4 (H_a^4): There exist significant differences between the accuracies of $\texttt{SentiStrength}^{O}$ and `SentiStrength*`.

The contingency matrix for the *McNemar's* test is presented in Table 4.4.4.2. As seen in the contingency matrix, `SentiStrength*` ($\mathcal{T}_b$) exhibits higher accuracies (compared to the

Table 4.15: Comparison of performances of **SentiStrength^O** and **SentiStrength***

Data	Sentiment	Met.	SentiStrength^O	SentiStrength*
Group-2	Positive	$\wp$	74.45%	**87.56%**
		$\Re$	**98.68%**	98.28%
		$\Finv$	84.87%	**92.61%**
	Negative	$\wp$	30.12%	**53.19%**
		$\Re$	97.66%	97.65%
		$\Finv$	46.04%	**68.87%**
	Neutral	$\wp$	96.69%	**97.94%**
		$\Re$	54.29%	**81.85%**
		$\Finv$	69.53%	**89.18%**
Group-3	Positive	$\wp$	26.00%	**40.44%**
		$\Re$	**86.90%**	82.01%
		$\Finv$	40.02%	**54.16%**
	Negative	$\wp$	47.13%	**69.10%**
		$\Re$	**76.77%**	72.65%
		$\Finv$	58.40%	**70.83%**
	Neutral	$\wp$	91.71%	91.22%
		$\Re$	58.02%	**79.54%**
		$\Finv$	71.07%	**84.98%**
Overall average accuracy		$\wp$	61.02%	**73.24%**
		$\Re$	78.72%	**85.33%**
		$\Finv$	68.74%	**78.82%**

Here, Note, `SentiStrength`^{O} uses the optimized dictionary
`SentiStrength`* uses our domain dictionary

Table 4.16: Contingency matrix for *McNemar's* test between `SentiStrength`^{O} and `SentiStrength`*

# of comments misclassified by both $\mathcal{T}_a$ and $\mathcal{T}_b$	942	140	# of comments misclassified by $\mathcal{T}_b$ but not by $\mathcal{T}_a$
# of comments misclassified by $\mathcal{T}_a$ but not by $\mathcal{T}_b$	**1,046**	3,436	# of comments correctly classified by both $\mathcal{T}_a$ and $\mathcal{T}_b$

Here, $\mathcal{T}_a$ = `SentiStrength`^{O} and $\mathcal{T}_b$ = `SentiStrength*`

SentiStrength^O) as $n_{10} > n_{01}$. The test obtains $p = 2.2 \times 10^{-16}$ where $p < \alpha$ and rejects our null hypothesis (H_0^4). Hence the alternative hypothesis (H_a^4) holds true indicating that the difference is statistically significant. The significantly superior accuracies of SentiStrength* implies that the domain dictionary we created for SentiStrength-SE outperforms the original SentiStrength's optimized dictionary. We, therefore, answer the research question RQ4 as follows:

Ans. to RQ4: *The domain dictionary we created for SentiStrength-SE performs significantly better than the optimized dictionary of the original SentiStrength.*

4.4.5 Comparison with a Large Domain-independent Dictionary

As mentioned before, domain difficulty is among the major reasons why domain-independent sentiment analysis techniques are found to have performed poorly when operated on in technical texts. This work of ours reveals the same as described in Section 4.2.5.2. For overcoming the domain difficulties, we have created domain-specific dictionary and heuristics in our SentiStrength-SE. However, compared to the existing domain-independent dictionaries available out there, our domain-specific dictionary is small in size with 167 positively and 310 negatively polarized entries. One might argue that a substantially *large* domain-independent dictionary might not suffer from the domain difficulties we are concerned about and may perform equally, if not better than our relatively small domain-specific dictionary. To verify this possibility, we compare the performances of our domain-specific dictionary with a large domain-independent dictionary. Particularly, we address the following research question:

RQ5: *Can a large domain-independent dictionary perform better than the domain-specific dictionary we created for SentiStrength-SE?*

4.4.5.1 Choosing a Domain Independent Dictionary for Comparison

There are several domain independent dictionaries (e.g., AFINN [118], MPQA [119], VADER [120], SentiWordNet [158], SentiWords [159], and the dictionary of Warriner et al. [160]) available for sentiment analysis in general. Islam and Zibran [57] compared the performances of AFINN [118], MPQA [119] and VADER [120] dictionaries in sentiment analysis of software engineering text. However, all those used dictionaries in the work of Islam and Zibran [57] can be considered to have *low* coverage. On the other hand, SentiWordNet [158], SentiWords [159], and the *extended* ANEW (Affective Norms for English Words) dictionary of Warriner et al. [160] are larger in size and have higher coverage compared to AFINN, MPQA and VADER dictionaries.

Among these three high coverage large dictionaries, we opt for the extended ANEW dictionary of Warriner et al. [160], which includes 13,915 English lemmas having 67% reported coverage [159].

Table 4.17: Conversion of valence scores from [+1,+9] to [-5,+5]

Score in [+1,+9]	+1	+2	+3	+4	+5	+6	+7	+8	+9
Score in [-5, +5]	-5	-4	-3	-2	+/- 1	+2	+3	+4	+5

We choose this dictionary for two main reasons: (i) this dictionary has already been used in software engineering studies [161, 58]; (ii) Use of parts-of-speech (POS) as context to determine words' polarities is found to show low accuracy in detecting sentiments in software engineering texts [57]. Therefore, we exclude `SentiWords` and `SentiWordNet` as these two dictionaries use POS as a context to determine words' polarities.

4.4.5.2 Range Conversion

In the extended `ANEW` dictionary of Warriner et al. [160], each word ω is assigned a valence score v_ω, which is a real number between +1.0 and +9.0 signifying the sentimental polarity and strength/intensity of the word ω. The sentimental polarity of the word ω, denoted as *sentiment*(ω), is interpreted according to Equation 4.5 below.

$$sentiment(\omega) = \begin{cases} Positive, & \text{if } v_\omega > +5.0 \\ Negative, & \text{if } v_\omega < +5.0 \\ Neutral, & \text{otherwise.} \end{cases} \tag{4.5}$$

In contrast, both the original `SentiStrength` and our `SentiStrength-SE` uses integer range [-5, +5] and a different interpretation for the same purpose. To use this extended `ANEW` dictionary in `SentiStrength`, we convert the valence score of each word in the extended `ANEW` dictionary from [+1.0, +9.0] range to [-5, +5] range. In doing that, the fractional value of v_ω is first rounded to its nearest integer $\hat{v}_\omega$. Then, using the conversion scale in Table 6.1, we convert each integer valence score $\hat{v}_\omega$ in the range [+1, +9] to $\mathcal{V}_\omega$ in the integer range [-5, +5]. For example, if the original valence score of a word rounded to the closest integer is +2, it is converted to -4, according to the mappings shown in Table 6.1. Such a conversion between ranges does not alter the original valence strength/intensity of the words [58]. A similar approach was adopted in a recent work [58] for range conversion of arousal scores.

4.4.5.3 Comparison between `SentiStrength`^{W} and `SentiStrength*`

We create another variant of the original `SentiStrength` by replacing its original dictionary with the one created based on the dictionary of Warriner et al., and call this new variant `SentiStrength`^{W}. `SentiStrength`^{W} is invoked for detecting sentiments in issue comments in Group-2 and Group-3 datasets. Then we compute the precision ($\wp$), recall ($\Re$), and F-score ($\Finv$) of `SentiStrength`^{W} in detection of each sentimental polarity. The computed metrics values for `SentiStrength`^{W} and `SentiStrength*` are presented side by side in Table 4.4.5.3.

Table 4.18: Comparison of performances of **SentiStrength**^{W} and **SentiStrength***

Data	**Sentiment**	**Met.**	**SentiStrength**^{W}	**SentiStrength***
Group-2	Positive	$\wp$	50.17%	87.56%
		$\Re$	**99.60%**	98.28%
		$\Finv$	66.73%	**92.61%**
	Negative	$\wp$	16.52%	**53.19%**
		$\Re$	85.94%	**97.65%**
		$\Finv$	27.71%	**68.87%**
	Neutral	$\wp$	91.11%	**97.94%**
		$\Re$	05.86%	**81.85%**
		$\Finv$	11.01%	**89.18%**
Group-3	Positive	$\wp$	10.40%	**40.44%**
		$\Re$	**96.79%**	82.01%
		$\Finv$	18.79%	**54.16%**
	Negative	$\wp$	28.33%	**69.10%**
		$\Re$	70.29%	**72.65%**
		$\Finv$	40.38%	**70.83%**
	Neutral	$\wp$	75.94%	**91.22%**
		$\Re$	08.23%	**79.54%**
		$\Finv$	14.84%	**84.98%**
Overall average accuracy		$\wp$	45.41%	**73.24%**
		$\Re$	61.12%	**85.33%**
		$\Finv$	52.10%	**78.82%**

Here, Note, `SentiStrength`^{W} uses the extended `ANEW` dictionary of Warriner et al. [160]
`SentiStrength`* uses our domain dictionary (created for `SentiStrength-SE`)

As seen in Table 4.4.5.3, in 16 of the 18 cases `SentiStrength`* achieves higher precision, recall, and F-score compared to those of `SentiStrength`^{W}. `SentiStrength`*'s recalls for positive sentiments only are slightly lower than those of `SentiStrength`^{W}. Notice that, for those same case, the `SentiStrength`^{W}'s *precision* is substantially lower compared to `SentiStrength`*. In every case, `SentiStrength`* maintains a nice balance between precision and recall in detecting sentimental and neutral comments. Such balancing between precisions and recalls results in higher F-scores for `SentiStrength`* in all cases. The overall accuracies, as presented in the bottom three rows of the table indicates significantly higher precision, recall, and F-score of `SentiStrength`* compared to `SentiStrength`^{W}. To determine the statistical significance of the observed differences in the accuracies, we perform *McNemar's* test between the results of `SentiStrength`^{W} and `SentiStrength`*. For the statistical test, we formulate our null and alternative hypotheses as follows:

Null Hypothesis-5 (H_0^5): There is no significant difference between the accuracies of `SentiStrength`^{W} and `SentiStrength`*.

Alternative Hypothesis-5 (H_a^5): There exist significant differences between the accuracies of `SentiStrength`^{W} and `SentiStrength`*.

Table 4.19: Contingency matrix for *McNemar's* test between `SentiStrength`^{W} and `SentiStrength`*

# of comments misclassified by both $\mathcal{T}_a$ and $\mathcal{T}_b$	1,014	68	# of comments misclassified by $\mathcal{T}_b$ but not by $\mathcal{T}_a$
# of comments misclassified by $\mathcal{T}_a$ but not by $\mathcal{T}_b$	**3,270**	1,212	# of comments correctly classified by both $\mathcal{T}_a$ and $\mathcal{T}_b$

Here, $\mathcal{T}_a$ = `SentiStrength`^{W} and $\mathcal{T}_b$ = `SentiStrength*`

The contingency matrix for the *McNemar's* test is presented in Table 4.4.5.3. As seen in the contingency matrix, `SentiStrength*` ($\mathcal{T}_b$) exhibits higher accuracies (compared to the `SentiStrength`^{W}) as $n_{10} > n_{01}$. The test obtains $p = 2.2 \times 10^{-16}$ where $p < \alpha$. Thus, the test rejects our null hypothesis (H_0^5). Hence the alternative hypothesis (H_a^5) holds true indicating that the differences in the accuracies of `SentiStrength*` and `SentiStrength`^{W} are statistically significant. Therefore, we answer the research question RQ5 as follows:

Ans. to RQ5: *For sentiment analysis in software engineering text, the domain-specific dictionary we created for `SentiStrength-SE` performs significantly better than a large domain-independent dictionary.*

4.4.5.4 Manual Investigation to Reveal Cause

We conduct an immediate qualitative investigation to reveal why the large dictionary of Warriner et al., having higher coverage, performs worse than our smaller domain-specific dictionary. We identify a set of comments C_m^W from Group-2 dataset, which are misclassified by `SentiStrength`^{W}. From the set C_m^W we distinguish another subset C_c^S, which are correctly classified by `SentiStrength`*. Then we randomly pick 50 comments from the set C_c^S for manual investigation.

From the manual investigation, we find the domain-specific variations in the meaning of words (i.e., the difficulty D_1 revealed in Section 4.2.5.2) as the main reason for the low accuracies of `SentiStrength`^{W}. For example, the following neutral comment is identified to have both negative and positive sentiments by `SentiStrength`^{W}.

> `"... crash for the same reason. Made some local fixes here."` (Comment ID: 149494)

The above comment is misclassified to have both positive and negative sentiments due to the presence of the words 'Crash' and 'Fix', which are negatively and positively polarized words respectively in the domain-independent `ANEW` dictionary of Warriner et al. [160]. In software engineering domain, both the words are neutral in sentiments. Due to the same reason, in detection of neutral comments, the performance of the dictionary of Warriner et al. is even worse than both the *optimized* and *default* dictionary of the original `SentiStrength`.

4.4.6 Comparison with an Alternative Domain Dictionary

Recall that our domain dictionary for `SentiStrength-SE` is developed using commit messages only. There is a possibility that a domain dictionary built on text from diverse sources may offer better performance. Thus, to verify this possibility, we create a second domain dictionary using text from diverse sources and compare this new alternative dictionary with the dictionary of `SentiStrenght-SE`. In particular, we address the following research question:

RQ6: *Can a domain dictionary developed using texts from diverse sources perform substantially better than `SentiStrength-SE`'s domain dictionary developed based on commit messages only?*

4.4.6.1 Building an Alternative Domain Dictionary

In addition to the 490 thousand commit messages used to develop `SentiStrength-SE's` dictionary, we obtain 1,600 *Code Review Comments* (CRC) [162], 1,795 *JIRA Issue Comments* (JIC) [58] and 4,423 Stackoverflow posts (SOP) [55]. We use software engineering texts from these diverse datasets to build the alternative domain dictionary.

Figure 4.3 depicts the steps/actions performed to develop this new dictionary. Here, in the first three steps (i.e., step-1 through step-3), we first produce a set S that includes all distinct words from all the four datasets. Then, we derive another set C_w of 1,198 words that are common in both S and the domain-independent dictionary of the original `SentiStrength`. These three steps are similar to

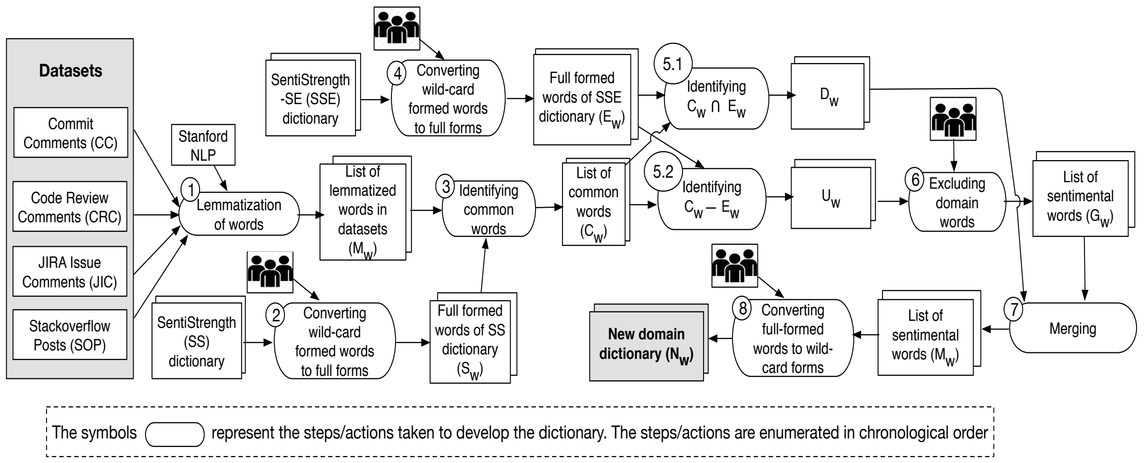

Datasets
Commit Comments (CC)
Code Review Comments (CRC)
JIRA Issue Comments (JIC)
Stackoverflow Posts (SOP)
Stanford NLP
1 Lemmatization of words
List of lemmatized words in datasets (M_w)
SentiStrength -SE (SSE) dictionary
4 Converting wild-card formed words to full forms
Full formed words of SSE dictionary (E_w)
3 Identifying common words
List of common words (C_w)
5.1 Identifying C_w ∩ E_w
D_w
5.2 Identifying C_w – E_w
U_w
6 Excluding domain words
List of sentimental words (G_w)
SentiStrength (SS) dictionary
2 Converting wild-card formed words to full forms
Full formed words of SS dictionary (S_w)
New domain dictionary (N_w)
8 Converting full-formed words to wild-card forms
List of sentimental words (M_w)
7 Merging
The symbols represent the steps/actions taken to develop the dictionary. The steps/actions are enumerated in chronological order

Figure 4.3: Procedural steps in developing a new alternative domain dictionary

the first three steps (Figure 4.1) in developing the domain dictionary of our `SentiStrength-SE` . However, here in step-1, we use four datasets instead of commit messages only.

In step-4, we transform the wild-card formed words in `SentiStrength-SE`'s dictionary to their full forms. The set of full-formed words is denoted as E_w. In step-5, we derive a set of words D_w from the set C_w such that D_w contains only those words that are common between C_w and E_w. Mathematically, $D_w = C_w \cap E_w$. We also derive another set U_w, which contains the words that are in C_w but not in E_w. Mathematically, $U_w = C_w$ - E_w.

All the words in D_w can safely be considered sentimental as those words are also present in the dictionary of `SentiStrength-SE`. We need to identify those words in U_w that are neutral in software engineering domain, but could be sentimental in general. Hence, in step-6, we involve three human raters (enumerated as A, B, and C) to independently identify those contextually neutral domain words. These human raters are the same three raters used in the development of the domain dictionary of `SentiStrength-SE`.

In Table 4.20, we present sentiment-wise percentage of cases where the human raters disagree pair-wise. We also measure the degree of inter-rater agreement in terms of *Fleiss-κ* [150] value. The obtained *Fleiss-κ* value 0.691 signifies substantial agreement among the independent raters. We consider a word as a neutral domain word when two of the three raters identify the word as neutral. Thus, 373 words are identified as neutral domain words, which we exclude from the set U_w resulting in another set of sentimental words G_w. Then, in step-7, by taking a union of the words in the sets G_w and D_w we form another set of words M_w, which contains all the sentimental words. Finally, in step-8, we adjust the words in M_w by reverting them to their wild-card forms (if available) to comply with `SentiStrength-SE`'s dictionary. This new set of words form our new domain dictionary (N_w). At this stage, we also make sure the words, which are manually added in the dictionary of `SentiStrength-SE`'s (see Section 4.3.1.1), are included in the set of words of the new domain dictionary. This alternative dictionary contains 225 positively and 495 negatively polarized sentimental entries.

4.4.6.2 Comparison between the New Dictionary and `SentiStrength-SE`'s Dictionary

We create a variant of `SentiStrength-SE` by replacing its domain dictionary with the newly created alternative domain dictionary. We call this variant `SentiStrength-SE`^{N}. Then we compare the performance of `SentiStrength-SE`^{N} against `SentiStrength-SE`, which actually implies a comparison of `SentiStrength-SE`'s dictionary with the *new* domain dictionary we have created.

Table 4.20: Inter-rater disagreements in interpretation of sentiments

	Disagreements between Human Raters		
Sentimental Polarity	**A, B**	**B, C**	**C, A**
Positive	11.32%	12.18%	12.22%
Negative	12.25%	11.19%	10.21%
Neutral	12.15%	10.53%	13.42%

Table 4.21: Comparison of performances of **SentiStrength-SE^N** and **SentiStrength-SE**

Data	**Sentiment**	**Met.**	**SentiStrength-SE^N**	**SentiStrength-SE**
Group-2	Positive	$\wp$	87.82%	**88.86%**
		$\Re$	98.81%	98.81%
		$\Finv$	92.99%	**93.57%**
	Negative	$\wp$	**53.45%**	53.42%
		$\Re$	**97.68%**	97.66%
		$\Finv$	**69.09%**	69.06%
	Neutral	$\wp$	98.13%	**98.14%**
		$\Re$	82.57%	**83.00%**
		$\Finv$	89.68%	**89.94%**
Group-3	Positive	$\wp$	39.16%	41.80%
		$\Re$	**82.62%**	82.04%
		$\Finv$	53.13%	**55.39%**
	Negative	$\wp$	**70.44%**	68.61%
		$\Re$	**72.25%**	71.00%
		$\Finv$	**71.33%**	69.78%
	Neutral	$\wp$	**90.71%**	90.64%
		$\Re$	78.94%	**80.05%**
		$\Finv$	84.42%	**85.02%**
Overall average accuracy		$\wp$	73.28%	**73.57%**
		$\Re$	**85.47%**	85.43%
		$\Finv$	78.91%	**79.06%**

Here, Note, `SentiStrength-SE`^{N} uses the newly created domain dictionary
`SentiStrength-SE` uses its own domain dictionary

We invoke `SentiStrength-SE`^{N} for detecting sentiments in issue comments in Group-2 and Group-3 datasets. Then we compute the precision ($\wp$), recall ($\Re$), and F-score ($\Finv$) of

SentiStrength^W in detection of each sentimental polarity. We present the computed metrics values obtained by SentiStrength-SE^N and SentiStrength-SE side by side in Table 4.4.6.2. As seen in Table 4.4.6.2, SentiStrength-SE^N performs slightly better than SentiStrength-SE in detection of negative sentiments. On the other hand, by observing precision and F-score values, we can say that SentiStrength-SE performs little better in detecting positive and neutral comments, although SentiStrength-SE^N achieves slightly higher recall values in those positive and neutral comments. The overall accuracies, as presented at the bottom three rows of Table 4.4.6.2, indicate that the performances between SentiStrength-SE^N and SentiStrength-SE do not differ substantially. To verify the statistical significance of the observed differences, we perform another *McNemar's* test between the results of SentiStrength-SE^N and SentiStrength-SE. For the statistical test, we formulate our null and alternative hypotheses as follows:

Null Hypothesis-6 (H_0^6): There is no significant difference between the accuracies of SentiStrength-SE^N and SentiStrength-SE.

Alternative Hypothesis-6 (H_a^6): There exist significant differences between the accuracies of SentiStrength-SE^N and SentiStrength-SE.

Table 4.22: Contingency matrix for *McNemar's* test between SentiStrength-SE^N and SentiStrength-SE

# of comments misclassified by both $\mathcal{T}_a$ and $\mathcal{T}_b$	1,033	49	# of comments misclassified by $\mathcal{T}_b$ but not by $\mathcal{T}_a$
# of comments misclassified by $\mathcal{T}_a$ but not by $\mathcal{T}_b$	83	4,399	# of comments correctly classified by both $\mathcal{T}_a$ and $\mathcal{T}_b$

Here, $\mathcal{T}_a$ = SentiStrength-SE^N and $\mathcal{T}_b$ = SentiStrength-SE

The contingency matrix for the *McNemar's* test is presented in Table 4.4.6.2. As seen in the contingency matrix, SentiStrength-SE^N ($\mathcal{T}_a$) and SentiStrength-SE ($\mathcal{T}_b$) exhibit almost equal accuracies as $n_{10} \approx n_{01}$. The test obtains $p = 0.0040$ where $p > \alpha$. Thus, the test fails to reject our null hypothesis (H_0^6). Therefore, we conclude that the performance of the newly created domain dictionary does not significantly differ from that of SentiStrength-SE's domain dictionary. We now formulate the answer to the research question RQ4 as follows:

Ans. to RQ6: *There is no statistically significant difference between the performances of the newly created domain dictionary and the domain dictionary of SentiStrength-SE.*

4.4.6.3 Manual Investigation to Determine Reasons

The result of the aforementioned comparison appear surprising to us, as we expected the newly created alternative dictionary to perform better than that of SentiStrength-SE. Recall that the newly created dictionary is larger than the domain dictionary of SentiStrength-SE.

SentiStrength-SE's dictionary has 167 positively and 310 negatively polarized sentimental words while the newly created one includes 225 positively and 495 negatively polarized words. While large number of entries in a dictionary can be helpful in achieving high recall, they can also misguide the sentiment analysis for a particular domain resulting in low precision. Hence, we manually investigate these possibilities in two phases and identify two reasons why the newly created alternative dictionary failed to outperform that of SentiStrength-SE.

Phase-1 Investigation: We randomly select a set of five issue comments for which SentiStrength-SE^N misclassifies their sentiments but SentiStrength-SE correctly classifies. One such comment is as follows:

```
"I disagree."
```
(Comment ID: 1787887_1)

SentiStrength-SE^N incorrectly identifies the above comment negatively emotional since the word 'disagree' recorded as a negatively polarized word in the newly created dictionary. SentiStrength-SE correctly identifies the comment as neutral as the word is not included in its dictionary. We identifies similar scenarios for all the five randomly picked issue comments.

Cause-1: Some emotional words present in the new domain dictionary also appear in many neutral comments of the ground-truth datasets. Which is why SentiStrength-SE^N ended up misclassifying those neutral comments as sentimental ones. This is a well-known problem of high coverage dictionaries [159].

Phase-2 Investigation: We randomly pick 20 inherently emotional words from the new domain dictionary, which are not present in the dictionary of SentiStrength-SE. Then, we search for those words in the ground-truth dataset and find five words (among the selected 20 words) (i.e., 'abhor', 'agony', 'appalling', 'crime' and 'delight') do not appear in any comments in the dataset.

Cause-2: This implies, although the new domain dictionary includes more sentimental words compared to the dictionary of SentiStrength-SE, many of those new sentimental words are unable to create any contributions in sentiment analysis due to their absence in the ground-truth datasets in use.

4.4.7 Evaluating the Contributions of Heuristics

In addition to the domain dictionary, SentiStrength-SE also includes a set of heuristics to guide the sentiment detection process towards higher accuracies. The heuristics, particularly designed for software engineering text, are also among the major contributions of this work. Here, we carry out a quantitative analysis to determine to what extent these heuristics contribute in the detection of sentiments in software engineering text. In particular, we address the following research question:

RQ7: *Do the heuristics integrated in `SentiStrength-SE` really contribute to improved sentiment analysis in software engineering text?*

We compare the performances of `SentiStrength-SE` and `SentiStrength*` to determine the contributions of the heuristics. Recall that `SentiStrength*` refers to the variant of the original `SentiStrength` that is forced to use our initial domain dictionary instead of its original one. Thus, `SentiStrength-SE` and `SentiStrength*` use the same domain dictionary and the only difference between them is the set of heuristics that are included in `SentiStrength-SE`. Hence, the heuristics are liable for any differences between the performances of `SentiStrength-SE` and `SentiStrength*`.

Table 4.23: Contributions of heuristics in `SentiStrength-SE`

Data	**Sentiment**	**Met.**	**SentiStrength-SE**	**SentiStrength***
Group-2	Positive	$\wp$	**88.86%**	87.56%
		$\Re$	**98.81%**	98.28%
		$\Finv$	**93.57%**	92.61%
	Negative	$\wp$	**53.42%**	53.19%
		$\Re$	**97.66%**	97.65%
		$\Finv$	**69.06%**	68.87%
	Neutral	$\wp$	**98.14%**	97.94%
		$\Re$	**83.00%**	81.85%
		$\Finv$	**89.94%**	89.18%
Group-3	Positive	$\wp$	**41.80%**	40.44%
		$\Re$	**82.04%**	82.01%
		$\Finv$	**55.39%**	54.16%
	Negative	$\wp$	68.61%	**69.10%**
		$\Re$	71.00%	**72.65%**
		$\Finv$	69.78%	**70.83%**
	Neutral	$\wp$	90.64%	**91.22%**
		$\Re$	**80.05%**	79.54%
		$\Finv$	**85.02%**	84.98%
Overall average accuracy		$\wp$	**73.58%**	73.24%
		$\Re$	**85.43%**	85.33%
		$\Finv$	**79.06%**	78.82%

*Here, `SentiStrength*` is forced to use our domain dictionary instead of its own one.

We present the performances of `SentiStrength-SE` and `SentiStrength*` in Table 4.4.6.3. For determining the effects of heuristics included in `SentiStrength-SE`, let us compare the right-most two columns in Table 4.4.6.3. We observe that the precision, recall, and F-score achieved by our `SentiStrength-SE` are consistently higher than those of `SentiStrength*` in most cases. In a few cases for the Group-3 dataset, `SentiStrength-SE`'s accuracy is nearly equal to those of `SentiStrength*`. The overall average accuracies, as presented at the bottom of Table 4.4.6.3, also indicate the superiority of our `SentiStrength-SE` over `SentiStrength*`, which implies that the heuristics incorporated in `SentiStrength-SE` really contribute to higher accuracy in the detection of sentiments in software engineering text.

However, as seen in Table 4.4.6.3, although the accuracy of `SentiStrength-SE` is higher compared to `SentiStrength*`, they do not differ by a large margin. Thus, it appears that the contributions of heuristics, in this particular case, are not substantial and unlikely to be statistically significant. To verify our observations, we perform a *McNemar's* test between the results of `SentiStrength*` and `SentiStrength-SE`. For the test, we formulate our null and alternative hypotheses as follows:

Null Hypothesis-5 (H_0^7): There is no significant difference between the accuracies of `SentiStrength-SE` and `SentiStrength*`.

Alternative Hypothesis-5 (H_a^7): There exist significant differences between the accuracies of `SentiStrength-SE` and `SentiStrength*`.

Table 4.24: Contingency matrix for *McNemar's* test between `SentiStrength-SE` and `SentiStrength*`

# of comments misclassified by both $\mathcal{T}_a$ and $\mathcal{T}_b$	993	78	# of comments misclassified by $\mathcal{T}_b$ but not by $\mathcal{T}_a$
# of comments misclassified by $\mathcal{T}_a$ but not by $\mathcal{T}_b$	**81**	4,433	# of comments correctly classified by both $\mathcal{T}_a$ and $\mathcal{T}_b$

Here, $\mathcal{T}_a$ = `SentiStrength-SE` and $\mathcal{T}_b$ = `SentiStrength*`

The contingency matrix for the *McNemar's* test is presented in Table 4.4.6.3. As seen in the contingency matrix, `SentiStrengthSE` ($\mathcal{T}_a$) and `SentiStrength*` ($\mathcal{T}_b$) exhibit almost equal accuracies as $n_{10} \approxeq n_{01}$. The test obtains $p = 0.874$ where $p > \alpha$. Thus, the test fails to reject our null hypothesis (H_0^7). Therefore, we conclude that the contribution of the heuristics in the tool is not significant in this particular case.

Based on our observations from the quantitative analysis and the statistical test, we now formulate the answer to the research question RQ5 as follows:

Ans. to RQ7: *Although the set of heuristics integrated in `SentiStrength-SE` contribute towards improved sentiment analysis in software engineering text, the perceived improvement is not statistically significant for the given datasets.*

Recall that, from the exploratory study (Section 4.2) using the Group-1 portion of the "Gold Standard" dataset, we found that the majority of the misclassifications of sentimental polarities are due to the limitations (difficulties D_1, D_3, D_6) of the dictionary in use (Table 4.3). Hence, the majority of misclassifications are to be corrected by using a domain dictionary, *leaving a relatively narrow scope for further contributions from the heuristics*, at least for this "Gold Standard" dataset. Our manual investigation of the datasets used in this study confirms the existence of *very few issue comments within operational scope of the heuristics*.

4.4.7.1 Further Manual Investigation

Although `SentiStrength-SE` is found to have performed better in most cases, as seen in Table 4.4.6.3, in four cases for the Group-3 dataset `SentiStrength-SE`'s accuracy is *marginally* lower than `SentiStrength`*. An immediate qualitative investigation reveals two reasons for this, which we discuss now.

First, our parameter setting to identify negations of sentimental words falls short in capturing negations in some cases. For example, in the following issue comment, the negation word "Don't" preceding the word 'Know' neutralizes the negatively polarized word 'Hell' due to the negation configuration parameter set to five in `SentiStrength-SE`.

> `"I don't know how the hell my diff program decide to add seemingly random CR chars, but i've removed them now"` (Comment ID: 306519_2)

A lower negation parameter could work better for this particular issue comment, but might perform worse for negations in others. Other possible solutions are discussed in Section 4.4.9.

Second, we also found instances of incorrect annotations for negative sentiments for some issue comments in the Group-3 dataset, which caused the accuracy of `SentiStrength-SE` appear to go down. Consider the following two issue comments.

> `"Inserting timestamps automagically would be bad because it would limit a whole swath of use cases"` (Comment ID: 1462480_2)

> `"If that would be the case, this would be bad design"` (Comment ID: 748115_2)

In the above two issue comments, the author stated merely the possibilities of negative scenarios that hadn't happened yet. These comments do not convey negative sentiments. But the human raters annotated with negative sentiments possibly due considering the use of negatively polarized word 'Bad' in the sentences.

These observations inspire us to conduct a deeper qualitative investigation of the success scenarios and especially the failure cases of `SentiStrength-SE` mainly to explore opportunities for further improvements to the tool. Hence, a qualitative evaluation of `SentiStrength-SE` is presented in the following section.

4.4.8 Qualitative Evaluation of SentiStrength-SE

Although from the comparative evaluations we found our `SentiStrength-SE` superior to the all selected tools, `SentiStrength-SE` is not a foolproof sentiment analysis tool. Indeed, 100% accuracy cannot be a pragmatic expectation. Nevertheless, we carry out another qualitative evaluation of `SentiStrength-SE` with two objectives: first, to confirm that the achieved accuracy found in the comparative evaluations did not occur by chance, and second, to identify failure scenarios and scopes for further improvements.

We first randomly pick 150 issue comments (50 positive, 50 negative, and 50 sentimentally neutral) from the Group-2 and Group-3 of the “Gold Standard" dataset for which `SentiStrength-SE` correctly detected the sentimental polarities. From our manual verification over these 150 issue comments, we are convinced that the design decisions, heuristics, and parameter configuration adopted in `SentiStrength-SE` have positive impacts on the accurate detection of sentimental polarities.

Next, we randomly choose another 150 issue comments (50 positive, 50 negative, and 50 sentimentally neutral) for which `SentiStrength-SE` failed to correctly detect the sentimental polarities. Upon manual investigation of those 150 issue comments, we find a number of reasons for the inaccuracies, a few of which are within the scope of the design decisions applied to `SentiStrength-SE`, and the rest falls beyond, which we discuss in Section 4.4.9. One of the reasons for failure is missing sentimental terms in our newly created domain dictionary. For example, `SentiStrength-SE` incorrectly identified the following comment as neutral in sentiment by misinterpreting the sentimental word ‘Stuck’ as a neutral sentimental word, since the word was not included in the dictionary, which we add to the dictionary of `SentiStrength-SE`’s release.

> `"For the first part, I got stuck on two points"` (Comment ID: 1610758_3)

Some other cases we have found inconsistencies in human rating of sentiments in issue comments, which are liable for inaccuracy in `SentiStrength-SE`. For example the following comment is rated as neutral in sentiment by human raters, although that contains the positive sentimental term ‘Thanks’ along with the exclamatory sign ‘!’.

> `"And many thanks to you Oliver for applying this so quickly!"`
> (Comment ID: 577184_1)

Our investigation reveals that 200 issue comments are wrongly interpreted in Group-3 by human raters that cause low accuracy in `SentiStrength-SE` for detecting positive sentiment, which is aligned with our earlier findings of such wrong interpretation of sentiments.

Although the additional preprocessing phase of `SentiStrength-SE` filters out unwanted content such as source code, URL, numeric values from the input texts, we found several instances where such contents escaped the filtering technique and misguided the tool.

In a few cases, we found that our heuristics to identify proper nouns fell short for not taking into account probable cases. For example, `SentiStrength-SE` incorrectly computed negative sentiment in the following issue comment. As seen in the following comment a developer thanked his colleague name ‘Harsh’.

```
"Thanks Harsh, the patch looks good ...  Since this is a new API, we
are not sure if want to change it.  Let's leave it as-is for the moment."
```
(Comment ID: 899420)

For failing to identify 'Harsh' as a proper noun, `SentiStrength-SE` considered the word sentimentally negative and erroneously detects negative sentiment in the message. Our immediate future plan includes further extension to our heuristics for locating proper nouns in text.

4.4.9 Threats to Validity

In this section, we discuss the threats to the validity of the empirical evaluation of `SentiStrength-SE` and our efforts in mitigating them.

4.4.9.1 Construct Validity and Internal Validity

Threats to construct validity relate to the suitability of the evaluation metrics. We use three metrics: *precision*, *recall* and *F-score* to evaluate the classification performances of `SentiStrength-SE` and other tools. All the three metrics have been commonly used for similar purposes in software engineering studies [162, 163, 55]. Only quantitative analysis may not portray the whole picture, which is why we have performed both quantitative and qualitative evaluation of `SentiStrength-SE`.

The accuracies in the computations of the metrics are subject to the correctness in the manual annotation of the issue comments with sentimental polarities. Hence, we have manually checked the annotations of issue comments in the "Gold Standard" dataset. We identified around 200 issue comments that are incorrectly labelled with wrong sentimental polarities. Nevertheless, we did not exclude those misclassified issue comments because they equally affect all the tools without favoring one over another.

To compare the performance of `SentiStrength-SE` against other tools (e.g., `SentiStrength`, `NLTK`, and `StanfordNLP`), we have used their default settings. Different settings of those tools might provide different results, however we adhered to their default settings due to their uses in earlier software engineering studies [34, 4, 41, 43, 40, 35, 36, 42, 48, 154, 164].

While optimizing `SentiStrength's` default dictionary for software engineering domain, we assigned three constant values +3, -3 and ± 1 to positive, negative and neutral comments, respectively. One may question the reasons for choosing those particular values instead of other values in the domains. For example, we could have used +2, +4 or +5 instead of +3 and -2,-4 or -5 instead of -3. Recall that those integer values not only indicate polarities of sentiments but also their intensities/strength [157]. In the computation of precision, recall and F-score, only the sentimental polarities are considered. Hence any value from +2 to +5 for positively polarized comments and any value from -2 to -5 for negatively polarized comments could be used in the process of optimization. We simply picked the values in the median, instead of choosing extreme ones at the domain boundaries.

We changed the range of valence scores of the words from [+1, +9] to [-5, +5] in the `ANEW` dictionary of Warriner et al. [160] to compare its performance against the domain dictionary of `SentiStrength-SE`. One might argue that the range conversion might have altered the original sentimental polarities of some words. We have considered this possibility and carefully designed the conversion scheme to minimize such possibilities. A random sanity check after the range conversion indicates absence of any such occurrence.

4.4.9.2 External Validity

The use of only one benchmark dataset (i.e., the “Gold Standard" dataset) can be considered a limitation of the empirical evaluation of our `SentiStrength-SE`. The outcome of the work could be more generalizable if more than one benchmark datasets could be used. At the time of first release of `SentiStrength-SE`, this “Gold Standard" dataset has been the only publicly available dataset especially crafted for the software engineering domain [52, 110]. A few newer datasets are available, but those are either not software engineering domain specific or they are even more specific to a narrower context (e.g., code review, product review). The issue comments in the benchmark dataset are collected from open-source systems and thus one may question whether or not the tools including ours will perform differently if applied on datasets drawn from industrial/proprietary projects. Producing a large dataset with human annotations is a tedious and time consuming task. We are working towards creating a second benchmark dataset for sentiment analysis in software engineering text. Once completed, we will release the dataset for the community.

Although there are diverse sources of textual content produced at different stages of software development and maintenance, the benchmark dataset we used contains only JIRA issue comments. Hence, one may argue that the results of the empirical comparison of tools might substantially vary if a dataset with a different type of text is used. Recall that the dictionary of our `SentiStrength-SE` tool is created based on commit comments. Thus, its superior performances on issue comments give us confidence that the tool will also perform well on other types of textual content.

4.4.9.3 Reliability

The methodology of this study including the procedure for data collection and analysis is documented in this chapter. The “Gold Standard" dataset [52] and all the tools (i.e., `SentiStrength-SE` [54], `SentiStrength` [45], `NLTK` [46] and `Stanford NLP` [146]) are available freely available online. Therefore, it should be possible to replicate the empirical evaluation of our tool.

4.5 Limitations of `SentiStrength-SE` and Future Possibilities

In this section, we discuss the limitations in the design and implementation of `SentiStrength-SE` as well as some directions for further improvements to our tool. To develop `SentiStrength-SE`, we have addressed the difficulties identified from the exploratory study described in Section 4.2. Still there are scopes for further improvements, as we also found from the qualitative evaluation of the tool. For example, we have observed in Section 4.4.8 that missing of sentimental words can mislead the `SentiStrength-SE`. We have created an alternative new domain dictionary for `SentiStrength-SE` using texts from diverse sources. Surprisingly, this newly created domain dictionary do not offer significant performance improvements. We plan to further extend our domain dictionary by integrating with different dictionary building approaches [163, 165, 166].

Our adopted approach for domain dictionary creation is unique from other attempted approaches [163, 165, 166]. We have deliberately chosen this approach for two reasons. First, we wanted to introduce a new approach, and second, it was not possible to adopt existing approached due to limitation of resources such as sentiment-annotated texts in software engineering [52]. Through the empirical evaluations, we have shown that our created domain dictionary is effective for sentiment analysis in software engineering. Moreover, in the process of development of the dictionary the identification of domain terms by three human raters might cause a subjectivity bias. However, we have measured the inter-rater agreements, and found reasonable agreements, which minimizes the threat substantially.

In the creation of our new domain dictionary, we used graduate students as the human raters, rather than expert software developers from industry. However, all the participants have at least three years of software development experience, which mitigates this threat. Moreover, it is reported that only minor differences exist between the performances of graduate students and professional software developers especially at small tasks involving simple judgements [167].

The use of only three human raters may be argued as a small number of participants. However, two to three raters have been common practice in successful software engineering studies [162, 163, 55, 168]. Moreover, through the empirical evaluations (both quantitative and qualitative), we have shown that our created domain dictionary is effective for sentiment analysis in software engineering text.

Several methods have been proposed and discussed to identify the scope of the negations of polarized words in sentiment analysis [169, 170], which can be applied to improve the performance of our negation handling approach. Many sophisticated approaches to identify the negation's scope include machine learning techniques [169, 171], complex rules [172], and identifying negated words using semantics of phrases [173]. However, many existing sentiment analysis approaches have relatively simple methods to identify scope of negation [125, 174]. Interestingly, performance of a negation detection method can be improved by domain adaptation [175]. In future, we will evaluate all the mentioned methods by applying in software engineering contexts to identify the best method to detect scope of negation.

Although our approach for filtering out code snippets may not correctly locate all code portions, but the filtering indeed minimizes them. Indeed, isolating inline source code from plain text content is a challenging task, especially when the text can have code written in diverse undeclared programming language. Such a code separation problem can be a separate research topic and limited scope attempts are made in the past [176]. We also plan to invest efforts along this direction to further improve `SentiStrength-SE`.

At this stage, we have not addressed the difficulties D_{10}, D_{11}, and D_{12}, which are included in our future plan. The detection of irony, sarcasm, and subtle emotions hidden in text is indeed a challenging research topic in NLP and not only related to software engineering texts. Even human interpretations of sentiments in text often disagree as such we also found in the "Gold Standard" dataset. Combining the dictionary-based lexical method with machine learning [177] and other specialized techniques [178] can lead to potential means to address these difficulties. We also plan to add to `SentiStrength-SE` the capability to identify interrogative sentences correctly mitigate the difficulty D_{10}.

4.6 Related work

To the best of our knowledge, the qualitative study (Section 4.2), is the first study that analyzes *public benchmark dataset* to expose the challenges to sentiment analysis in software engineering. And, we have developed the first sentiment analysis tool, `SentiStrength-SE`, crafted especially for software engineering domain, which we expect to produce superior performance in other technical domains as well.

Aside from our tool, there are only four prominent tools/toolkits namely, `SentiStrength` [45], `Stanford NLP` [146], `NLTK` [46], and `Alchemy` [179], which facilitate automatic sentiment analysis in plain texts. The first three of these tools have been used for sentiment analysis in software engineering domain, while `SentiStrength` is used most frequently in the studies as presented in Table 4.6. We categorize those studies for better understanding of the uses of those tools and the contributions of those studies in software engineering domain. Those tools, which are previously used in software engineering area, but *not for sentiment analysis*, are excluded from the table. Notably, none of the studies used any domain specific tool to detect sentiments.

`Alchemy` [179] is a commercial toolkit that offers limited sentiment analysis as a service through its published APIs. According to the study of Jongeling et al. [142] the performance of `Alchemy` is lower than `SentiStrength` [45] and `NLTK` [46]. `NLTK` and `Stanford NLP` [146] are general purpose natural language processing (NLP) library/toolkit, which expect the user to have some NLP background and to write scripting code for carrying out sentiment analysis in plain text. In contrast,

Table 4.25: Uses of (domain-independent) tools for *sentiment analysis* in software engineering

Tools	Type of Work	Uses in Software Engineering Research
`SentiStrength` [45]	Analyzing sentiments in software engineering (SE)	[34, 43, 37, 38, 36, 50, 180, 181]
	Applications of sentiments in SE	[4, 40, 35, 42, 51, 182, 136]
	Benchmarking study	[48, 142]
`NLTK` [46]	Analyzing sentiments in SE	[41, 164]
	Benchmarking study	[48, 142]
`Stanford NLP` [146]	Applications of sentiments in SE	[154]
	Benchmarking study	[48, 142]

`SentiStrength` is a dedicated tool that applied a lexical approach for automated sentiment analysis and is ready to operate without needing to write any scripting code (for natural language processing). Perhaps, these are among the reasons why, in software engineering community, `SentiStrength` has gained popularity over the alternatives. The same reasons also made us choose this particular tool as the basis of our work. Our `SentiStrength-SE` reuses the lexical approach of the original `SentiStrength` and is also ready to be used off the shelf.

All of the aforementioned four tools (i.e., `SentiStrength` [45], `Stanford NLP` [146], `NLTK` [46], and `Alchemy` [179]) are developed and trained to operate on non-technical texts drawn from social interactions, web pages, and they do not perform well enough when operated in a technical domain such as software engineering. Domain-specific (e.g., software engineering) technical uses of inherently emotional words seriously mislead the sentiment analyses of those tools [41, 43, 48, 50] and limit their applicability in software engineering area. We have addressed this issue by developing the first software engineering domain-specific dictionary included in our tool `SentiStrength-SE`. Along this direction Mäntylä et al. [39] developed a dictionary to capture *emotional arousal* in the software engineering texts.

Apart from creating domain dictionary, a variety of machine learning (ML) techniques such as, Naive Bayes classifier (NB), Support Vector Machine (SVM) [125], and Logistic Regression (LR) [183] have been explored in an attempt to minimize the domain difficulty. However, the performances of all these three classifiers are reported lower when operated on domain-specific texts [184]. Nonetheless, recently Murgia et al. [185] applied several ML techniques (e.g., NB, SVM) to identify emotions *love*, *joy* and *sadness* only in contrast to our tool `SentiStrength-SE` that can differentiate *positivity*, *negativity* and *neutrality* of software engineering texts. [1]Again, Panichella et al. [168] [1]used NB classifier to detect sentiments in software users' reviews. However, the accuracy of their classifier was not reported. Those two tools are not publicly available to compare against our tool `SentiStrength-SE`. Moreover, we avoided to apply ML technique to implement

`SentiStrength-SE` due to [1]the limitations of ML for sentiment analysis that include its difficulty to integrate into a classifier and learned models often have poor adaptability between different text genres or domains as they often rely on domain specific features found in their training data [184].

Blaz and Becker [163] proposed three almost equally performing methods, a *Dictionary Method (DM)*, a *Template Method (TM)* and a *Hybrid Method (HM)* for sentiment analysis in "Brazilian Portuguese" texts in IT (Information Technology) job submission tickets. The DM is a pure lexical approach similar to that of our `SentiStrength-SE`. Although their techniques might be suitable for formally structured texts, those may not perform well in dealing with informal texts that are frequently used in software engineering artifacts such as commit comments. In contrast, from the empirical evaluation over commit comments, our `SentiStrength-SE` is found to have high accuracy in detecting sentiments in those informal software engineering texts. The proposed methods of Blaz and Becker [163] are developed and evaluated against text written in "Brazilian Portuguese" language instead of English. Thus, their approach and reported results are not directly comparable to ours.

Similar to the qualitative study included in our work, Novielli et al. [50] also conducted a relatively brief study of the challenges against sentiment analysis in "social programmer ecosystem". They also used `SentiStrength` for the detection of emotional polarities and reported only domain difficulty as a key challenge. In their work, they manually studied only 100 questions and their follow-up comments as well as 100 answers and their follow-up discussions obtained from Stack Exchange Data Dump [186]. In contrast, based on a deeper analysis over a publicly available benchmark dataset, our study exposes 12 difficulties including the domain dependency. In addition, we address a portion of those difficulties and develop a domain-specific tool for improved sentiment analysis in software engineering text.

Our `SentiStrength-SE` is the *first* software engineering domain specific sentiment analysis tool. Soon after the release of `SentiStrength-SE`, four domain-specific tools/toolkits (i.e., `Senti4SD` [55], `SentiCR` [162], `EmoTxt` [56], and `SentiSW` [187]) have appeared over the last few months. Similar to our `SentiStrength-SE`, `Senti4SD` is also a software engineering domain specific sentiment analysis tool. `Senti4SD` applies machine learning based on lexicon and keyword based features for detecting sentiments. `SentiSW` also applies machine learning techniques to detect sentiments at *entity-levels*. `EmoTxt` [56] is an open-source toolkit for detecting six basic emotions (i.e., love, joy, anger, sadness, fear, and surprise) from *technical text*. The authors of `SentiCR` declared the scope of this tool limited to code review comments only. Again, the applicability of `SentiSW` is limited to only JIRA issue comments. The reported scope of `EmoTxt` is technical domain, which is wider than software engineering domain. On the contrary, the scopes of `SentiCR` and `SentiSW` are limited to narrower domains of code review comments and JIRA issue comments, respectively.

Recently, two separate studies have been conducted to compare the performances of those domain specific sentiment analysis tools. In the first study, Islam and Zibran [188] compared the performances of SentiStrength-SE, Senti4SD and EmoTxt and found no convincing winner among the tools for sentiment analysis in software engineering. In the later study, Novielli et al. [189] compared the performances of SentiStrength-SE, Senti4SD and SentiCR and found the unsupervised approach of SentiStrength-SE had provided comparable performance to that of supervised techniques (i.e., Senti4SD and SentiCR). SentiSW was not available at the time of conducting the two comparative studies. However, developers of SentiSW compared its performance against SentiStrength-SE and found their tool's superiority over SentiStrength-SE in detecting sentiments expressed in JIRA issue comments. While all those studies compared the performances of domain specific tools, for the first time, Jongeling et al. [48, 142] compared the performances of domain independent tools SentiStrength [45], NLTK [46], Stanford NLP [146] and Alchemy [179] to measure their applicability in software engineering domain.

We do not claim SentiStrength-SE to be the best tool among all the few tools available for sentiment analysis in software engineering text. Instead, by developing and evaluating the *domain-specific* SentiStrength-SE, we demonstrate that, for sentiment analysis in software engineering text, a *domain-specific* technique performs significantly better compared to its *domain-independent* counterparts. As mentioned before, our SentiStrength-SE is the first domain-specific tool for sentiment analysis in software engineering. Other researches might have taken inspiration from our work [110] in attempting domain-specific solutions resulting in several domain-specific tools discussed above.

4.7 Summary

We have first presented an in-depth qualitative study to identify the difficulties in automated sentiment analysis in software engineering texts. Among the difficulties, the challenges due to domain dependency are found the most dominant. To address mainly the *domain difficulty*, we have developed a domain-specific dictionary especially designed for sentiment analysis in software engineering text.

We also develop a number of heuristics to address some of the other identified difficulties. Our new domain dictionary and the heuristics are integrated in SentiStrength-SE, a tool we have developed for improved sentiment analysis in textual contents in a technical domain, especially in software engineering. Our tool reuses the lexical approach of SentiStrength [45], which, in software engineering, is the most widely adopted sentiment analysis technique. Our SentiStrength-SE is the first domain-specific sentiment analysis tool especially designed for software engineering text.

Over a large dataset (i.e., Group-2 and Group-3) consisting of 5,600 issue comments, we carry out quantitative comparisons of our *domain-specific* SentiStrength-SE with the three most popular *domain independent* tools/toolkits (i.e., `NLTK` [46], `Stanford NLP` [47], and the original `SentiStrength` [45]. The empirical comparisons suggest that our domain-specific `SentiStrength-SE` is [1]significantly superior to its *domain independent* counterparts in detecting emotions in software engineering textual contents.

Using both quantitative and qualitative evaluations, we also separately verify the effectiveness of the design decisions including the domain dictionary and heuristics we have included in our *domain-specific* `SentiStrength-SE`. From the evaluations, we found that our newly created domain dictionary makes statistically significant contributions to improved sentiment analysis in software engineering text. However, the heuristics we developed to minimize the issues beyond the domain difficulties are found not to have substantial impacts on sentiment analysis of the chosen datasets. The non-substantial impact of the heuristics further validates that the improvements in the accuracies of `SentiStrength-SE` are attributed to its being *domain-specific*. Thus, we demonstrate that, for sentiment analysis in software engineering text, a domain-specific technique performs substantially better than domain independent techniques.

Chapter 5

Comparison of Sentiment Analysis Tools

In the last chapter (Chapter 4), we identified the difficulties for sentiment analysis in software engineering and describe the development procedure of our sentiment analysis tool `SentiStrength-SE` to overcome most of the identified difficulties. Recent attempts have led to the development of a few domain specific sentiment analysis tools especially designed to deal with software engineering text. In this chapter, we compare the performances and agreements exist among the software engineering domain specific sentiment analysis tools in detecting sentiments.

This chapter is organized as follows. In Section 5.1, we describe the motivation and context of the work. We describe the datasets and domain specific sentiment analysis tools that we use in this work respectively in Section 5.2 and Section 5.3. Tools' performance evaluations and findings are illustrated in Section 5.4. Threats to validity of the work are described in Section 5.5. We discuss the related work in Section 5.6. Finally, Section 5.7 concludes the chapter.

5.1 Introduction

Sentiment Analysis (SA) in software engineering (SE) text has recently drawn interests in the community [37, 38]. Earlier attempts used general-purpose (i.e., domain independent) sentiment detection tools (e.g., `SentiStrength` [45], `NLTK` [46] and `Stanford NLP` [146]) for SA in SE text. Those general purpose SA tools are found to have very low accuracies when operated on text from a technical domain such as software engineering [37, 48, 43]. Those SA tools were developed and trained using data from non-technical social networking media (e.g., twitter posts, forum posts, movie reviews) and perform poorly for software engineering text largely due to domain specific variations in meanings of frequently used technical terms [110].

Thus, recent attempts have led to the development of a few domain specific SA tools especially designed to deal with SE text. Each of these domain specific SA tools were originally evaluated using a different dataset and compared against the existing domain independent SA tools. The datasets (e.g., JIRA issue comments, Stack Overflow posts, code review comments) differ in the proportion and category of technical text they include. The accuracies of these SE domain specific tools have never been compared using multiple datasets.

Using multiple datasets, we carry out a quantitative comparison of three recently released SE domain specific SA tools. In our study, we address the following two research questions.

RQ1: *Can we identify a tool, which shows the highest accuracy across different datasets*? — We investigate which tool achieved higher accuracy in which dataset, and we distinguish a tool which achieves overall the best accuracy across all the datasets. This will help one in choosing the most appropriate tool for SA in SE text.
RQ2: *To what extent do the different sentiment analysis tools (dis)agree with each other*? —Here we examine to what extent the SA tools (dis)agree on their detection of sentimental polarities (i.e., positivity, negativity, and neutrality) in SE text. This agreement analysis will help in identifying the spots where those tools might need improvements.

5.2 Datasets

In our study, we use three ground-truth datasets drawn from software development ecosystems. A summary of these three datasets is presented in Table 5.1.

Table 5.1: Summary of the Datasets Used in this Study

Dataset	Group	# of Comments				
		Total	Pos	Neg	Neu	Non-neg
JIRA Issue Comments	Group-2	1,576	748	128	700	1,448
	Group-3	4,000	375	672	2,953	3,324
SOP	NA	4,423	1,527	1,202	1,694	3,221
CRC	NA	2,000	NA	398	NA	1,202

5.2.1 JIRA Issue Comments (JIC) Dataset

This dataset is based on the work of Ortu et al. [52]. The entire dataset is divided in three groups named as Group-1, Group-2, and Group-3. Group-1 contains 392 issue comments and Group-2 contains 1,600 issue comments. Group-3 contains 4,000 sentences written by developers. Each individual text (i.e., issue comments and sentences) in the dataset are manually annotated with emotions such as *love, joy, surprise, anger, sadness* and *fear*. For manual annotation, each of the 5,992 individual text is interpreted by n distinct human raters [52] and annotated with emotional expressions as found in those comments. For Group-1, $n = 4$ while for Group-2 and Group-3, $n = 3$. In this study, we use the Group-2 and Group-3 portions of the dataset as those were also used in other studies [110, 52].

5.2.1.1 Emotional Expressions to Sentimental Polarities

We compute sentimental polarities (i.e., positivity, negativity, and neutrality) from the emotional expressions (i.e., *love, joy, surprise, anger, sadness, fear*) as follows. Emotional expressions joy and $love$ denote *positive* sentiment, while $anger$, $sadness$, and $fear$ indicate *negative* sentiment. We take special measurement for the *surprise* expressions as in some cases, an expression of *surprise* can indicate positive polarity, denoted as $surprise^{+}$, while in other cases it can express a *negative* sentiment, denoted as $surprise^{-}$. Thus the issue comments in the benchmark dataset, which are annotated with *surprise* emotion, need to be further classified based on the sentimental polarities they convey. Hence, we get each of such comments interpreted by three human raters (computer science graduate students), who independently assign polarities of the surprise expressions in each comments.

We consider a *surprise* expression in a comment negatively polarized (or positively), if two of the three raters identify negative (or positive) polarity in it. We found 79 issue comments in the benchmark dataset, which were annotated with the *surprise* expression. 23 of them express *surprise* with positive polarity and the rest 56 convey negative *surprise*.

Then we split the set $\mathcal{E}$ of emotional expressions into two disjoint sets. The first set is $\mathcal{E}_{+} = \{joy, love, surprise^{+}\}$ and the second set is $\mathcal{E}_{-} = \{anger, sad, fear, surprise^{-}\}$. Thus, $\mathcal{E}_{+}$ contains only the positive sentimental expressions and $\mathcal{E}_{-}$ contains only the negative sentimental expressions. A similar approach is also used in other work [110, 48, 142] to categorize emotional expressions according to their polarities.

5.2.1.2 Assignment of Sentiments to Text

A piece of text is assigned positive sentiment if maximum number of raters among the n raters identify positive sentiment in that post. For example, in Group-2, a text $\mathcal{T}$ is considered to have a positive sentiment, if two of the three raters agree on perceiving positive sentiment in $\mathcal{T}$. Similarly, negativity and neutrality of pieces of text are also determined based on majority agreements. We find that human raters could not agree on the sentiments of 24 issue comments in Group-2. These 24 comments are excluded from this study.

5.2.2 Stack Overflow Posts (SOP) Dataset

The second ground-truth dataset we use is based on the work of Calefato et al. [128]. This dataset is composed of 4,423 posts from Stack Overflow. Each if the 4,423 posts is interpreted and annotated with sentimental polarities (i.e., positive, negative, neutral) by three distinct human raters and a total 12 different raters were used to annotate the entire dataset. The sentiments expressed in a particular post are determined based on majority agreements. Thus, in this dataset, 35% of posts convey positive sentiment and 27% express negative sentiment while 38% of posts are neutral in sentiments.

5.2.3 Code Review Comments (CRC) Dataset

The third dataset used in this work is based on the work of Ahmed et al. [111]. This dataset contains manually annotated 2,000 code review comments drawn from twenty open-source projects. Three human raters independently label each of the 2,000 code review comments as *positive*, *negative* or *neutral* in accordance with the sentimental polarities they perceive in the comment. The decisive sentiment of a particular comment is determined based on majority agreements. Thus, Ahmed et al. produced a three-class dataset consisting of comments with positive, negative, or neutral sentiments. However, in their publicly published dataset, the positive and neutral comments are merged in one *non-negative* class. Therefore, in this work, we have to use this two-class dataset where 19.9% comments express negative sentiments and the rest 80.1% comments convey non-negative sentiments.

5.3 Sentiment Analysis Tools under Study

We study the following three SE domain specific SA tools released in last couple of years.

SentiStrength-SE: The tool `Sentistrength-SE` [110] is the first domain specific tool especially developed for sentiment analysis in software engineering text. Given piece of text $\mathcal{T}$, `SentiStrength-SE` computes a pair $\langle p_c, n_c \rangle$ of integers, where $+1 \leq p_c \leq +5$ and $-5 \leq n_c \leq -1$. Here, p_c and n_c respectively represent the positive and negative sentimental scores for the given text $\mathcal{T}$. In `Sentistrength-SE`, a given text $\mathcal{T}$ is considered to have positive sentiment if $p_c > +1$. Similarly, a text is held containing negative sentiment when $n_c < -1$. Besides, a text is considered sentimentally neutral when the sentimental scores for the text appear to be $\langle 1, -1 \rangle$.

Senti4SD: The tool `Senti4SD` [128] is a machine learning based tool specifically trained to support sentiment analysis in software engineering related text. By exploiting a suite of both lexicon and keyword-based features, it can detect positive, negative, and neutral sentiments in text. The authors of the classifier claim that it reduces the misclassifications of neutral and positive posts.

EmoTxt: The tool `EmoTxt` [56] is an open-source toolkit that can detect a set of six basic emotions, namely *love, joy, anger, sadness, fear*, and *surprise* from technical text. To convert the emotional expressions to sentimental polarities we use the same procedure as applied to the JIC dataset as described earlier in Section 5.2.1. However, `EmoTxt` cannot identify the polarities of *surprise* expression. To mitigate this issue, from all the datasets, we exclude those comments, which are identified to convey only *surprise* expression by `EmoTxt` (elaborated later in Section 5.4.1).

5.4 Evaluation and Findings

To address the first research question (as mentioned in Section 5.1), we perform a two-stage analysis of comparative accuracies of the sentiment analysis tools. The second research question is addressed through an agreement analysis, as described in Section 5.4.2.

5.4.1 Comparative Accuracy Analysis

The accuracy of sentiment detection is measured in terms of *precision*, *recall*, and *F-score* separately computed for each of the three sentimental polarities (i.e., positivity, negativity and neutrality). Given a set $\mathcal{S}$ of textual contents, *precision* (p), *recall* (r), and *F-score* ($\mathrel{\reflectbox{F}}$) for a particular sentimental polarity e is calculated as follows:

$$p = \frac{|\ \mathcal{S}_e \cap \mathcal{S}_e^t\ |}{|\ \mathcal{S}_e^t\ |}, \quad r = \frac{|\ \mathcal{S}_e \cap \mathcal{S}_e^t\ |}{|\ \mathcal{S}_e\ |}, \quad \mathrel{\reflectbox{F}} = \frac{2 \times p \times r}{p + r}$$

where $\mathcal{S}_e$ represents the set of texts having sentimental polarity e (according to ground-truth), and $\mathcal{S}_e^t$ denotes the set of texts that tool t detects to have the sentimental polarity e.

Stage-1 Evaluation: We separately operate the three tools (`SentiStrength-SE`, `Senti4SD`, and `EmoTxt`) on the JIC (JIRA Issue Comments) dataset and the SOP (Stack Overflow Posts) dataset. We find 302 comments in JIC dataset and 282 posts in the SOP dataset, for which `EmoTxt` finds *surprise* expression only and cannot proceed further to determine polarities of those *surprise* expression. These comments and posts are excluded from the respective datasets to maintain a level-playing field for all the tools.

For each of the three sentimental polarities (i.e., positivity, negativity, and neutrality), we compare the tools' outcome with the respective ground-truth and separately compute precision, recall, and F-score for all the tools for both JIC and SOP datasets. Table 5.2 presents the accuracies (in precision, recall, and F-score) of the of the three tools in their detection of *positive*, *negative* and *neutral* sentiments. The average overall accuracies are presented at the bottom three rows. The highest metric values are highlighted in bold.

As seen in Table 5.2, the accuracy of `EmoTxt` has remains lower than `SentiStrength-SE` and `Senti4SD` for both the datasets. `SentiStrength-SE` achieves the highest accuracy for the JIC dataset while `Senti4SD` achieves the highest accuracy for the SOP dataset. The overall average accuracies also indicate that both `SentiStrength-SE` and `Senti4SD` perform better than `EmoTxt` by a considerable margin. Although it is difficult to distinguish a clear winner, `SentiStrength-SE` can be held superior to `Senti4SD` due to its overall higher recall and slightly higher F-score.

It is interesting to observe that `Senti4SD` and the SOP dataset are from the same authors. Similarly, the JIC dataset is the same dataset on which `SentiStrength-SE` was originally evaluated at the time of its release. Hence, there is a chance of bias and it is worth evaluating all these tools using a dataset on which none of the tools are ever been tested. We carry out such an evaluation in stage-2 using the third dataset described earlier in Section 5.2.3.

Stage-2 Evaluation: We separately operate all the three tools on the CRC (Code Review Comments) dataset. Again, we find 188 comments, which `EmoTxt` identified to express *surprise* emotion only. Similar to the stage-1 evaluation, we exclude these 188 comments from our analysis. Then for non-

Table 5.2: Tools' Accuracies for JIC Dataset and for SOP Dataset

Dataset	Sentiment	Metrics	SentiStrength-SE	Senti4SD	EmoTxt
JIRA Issue Comments	Pos	*p*	**64.73%**	54.04%	61.26%
		r	**94.29%**	75.20%	70.12%
		Ⅎ	**76.76%**	62.89%	65.39%
	Neg	*p*	**70.96%**	54.78%	34.34%
		r	**78.14%**	41.56%	61.74%
		Ⅎ	**70.91%**	47.26%	44.13%
	Neu	*p*	**91.87%**	80.85%	88.92%
		r	**79.63%**	74.49%	65.98%
		Ⅎ	**85.31%**	77.54%	75.75%
Stack Overflow Posts	Pos	*p*	82.69%	**97.44%**	88.57%
		r	94.15%	**97.44%**	94.15%
		Ⅎ	88.05%	**97.44%**	91.27%
	Neg	*p*	73.45%	**93.03%**	64.93%
		r	78.17%	95.94%	**96.02%**
		Ⅎ	75.74%	**94.46%**	77.47%
	Neu	*p*	80.73%	**97.29%**	94.84%
		r	69.10%	**94.78%**	51.15%
		Ⅎ	74.47%	**96.02%**	66.46%
Overall average accuracy		*p*	77.41%	**79.57%**	72.14%
		r	**82.24%**	79.90%	73.19%
		Ⅎ	**79.75%**	79.73%	72.66%

negative and negative sentimental texts, we separately compute precision, recall, and F-score for all the three tools. The computed accuracy measurements are presented in Table 5.3.

Table 5.3: Tools' Accuracies for Code Review Comments Dataset

Dataset	Sentiment	SentiStrength-SE	Senti4SD	EmoTxt
Code Review Comment	Non-neg	**83.64%**	81.69%	81.45%
		92.67%	**93.13%**	82.42%
		87.92%	87.03%	81.93%
	Neg	50.23%	55.09%	**84.95%**
		34.69%	28.75%	24.69%
		41.04%	37.78%	38.26%
Overall average accuracy		66.94%	68.39%	**83.20%**
		63.68%	60.94%	53.56%
		65.26%	64.45%	65.16%

As seen in Table 5.3, all the tools appear to have performed much better in the detection of non-negative sentiments compared to their accuracies in the detection of negative sentiments. `EmoTxt` achieves the highest precision in detecting negative sentiments, `Senti4SD` has the highest recall in the detection of non-negative sentiments. On the other hand, `SentiStrength-SE` achieves the higher precision and F-score for non-negative sentences as well as the highest recall and F-score for negative sentiments.

The overall average accuracies suggest that `EmoTxt` has the highest precision but the lowest recall and the differences from those the other tools are substantial. On the contrary, `SentiStrength-SE` achieves the higher recall and F-score but the lowest precision. The overall average precision of `Senti4SD` is slightly higher than that of `SentiStrength-SE`, but `Senti4SD`'s recall and F-score are lower by small margin than those of `SentiStrength-SE`. Thus, similar to the result of stage-1 evaluation, `SentiStrength-SE` can be considered to have achieved slightly better accuracies than the other tools, in terms of recall and F-score, although the differences can be perceived negligible.

Based on our observations and analyses of results in both stage-1 and stage-2 evaluations, we now derive the answer to the first research question (RQ1) as follows.

Ans. to RQ1: *Accuracies of the tools vary across datasets and sentiments. None of the tools stand out as substantially superior to the other tools. However, `SentiStrength-SE` consistently achieves the highest recall with competitive precision across sentiments and datasets.*

5.4.2 Analysis of Agreements

For addressing the second research question (RQ2), we perform an agreement analysis over the tools sentiment detection results. We compute P_{xy}^{e} denoting the agreement between tool x and tool y for a particular sentiment e as follows:

$$P_{xy}^{e} = \frac{\mid S_e^x \cap S_e^y \mid}{\mid S_e \mid} * 100$$

Table 5.4: Agreements between Tool-pairs in the Detection of Sentiments

Dataset	**Sentiment**	**SSE* vs. Senti4SD**	**SSE* vs. EmoTxt**	**Senti4SD vs. EmoTxt**
JIRA Issue Comments	Pos	**77.88%**	72.17%	63.51%
	Neg	51.32%	**72.03%**	49.74%
	Neu	**81.86%**	72.29%	70.00%
Stack Overflow Posts	Pos	**94.35%**	92.31%	93.63%
	Neg	79.02%	80.71%	**93.49%**
	Neu	**82.32%**	83.22%	81.26%
Code Review Comments	Non-neg	**94.70%**	86.05%	83.81%
	Neg	65.00%	66.56%	**66.88%**

SSE* = SentiStrength-SE

The computed agreements between each *pair* of the tools are presented in Table 5.4. It can be observed that the agreements between the tools vary across different datasets and sentiments. For example, the tools `SentiStrength-SE` and `Senti4SD` show the highest agreement (94.70%) for non-negative sentiments in the CRC (Code Review Comments) dataset whereas the lowest agreement (49.74%) is found between `EmoTxt` and `Senti4SD` in the detection of negative sentiments in the JIC (JIRA Issue Comments) dataset.

We observe two patterns in the agreements of the tools presented in Table 5.4. First, `SentiStrength-SE` and `Senti4SD` always achieve the highest agreement for non-negative (i.e., positive and neutral) sentiments. Second, for each pair of tools, on every dataset, the lowest agreement is found in the detection of negative sentiments. The only exception to this holds for `Senti4SD` and `EmoTxt` in the SOP (Stack Overflow Posts) dataset.

Table 5.5: Agreements among Tool-trio in the Detection of Sentiments

Dataset	**Sentiment**	**Fleiss' κ**	**Agreement Strength**	**Reason of Interpretation**
JIRA Issue Comments	Pos	0.108	poor	$0.00 \leq \kappa \leq 0.19$
	Neg	0.087	poor	$0.00 \leq \kappa \leq 0.19$
	Neu	0.101	poor	$0.00 \leq \kappa \leq 0.19$
Stack Overflow Posts	Pos	0.498	moderate	$0.40 \leq \kappa \leq 0.59$
	Neg	0.164	poor	$0.00 \leq \kappa \leq 0.19$
	Neu	0.731	substantial	$0.60 \leq \kappa \leq 0.79$
Code Review Comments	Non-neg	0.462	moderate	$0.40 \leq \kappa \leq 0.59$
	Neg	0.265	fair	$0.20 \leq \kappa \leq 0.39$

To further examine the agreements in the tool-trio (i.e., among the three tools), we compute Fleiss' kappa [150] (adaptation of Cohen's kappa for three or more participants), denoted as κ, for each sentiment in all the three datasets. The computed Fleiss' kappa (κ) values and their interpretations are presented in Table 5.5. As seen in the table, there are only one instances of each 'fair' and 'substantial' agreements, two instances of 'moderate' agreements and in all other cases, there are 'poor' agreements among the tool-trio. The tools agree the least in the JIC dataset.

In every dataset, the lowest Fleiss' kappa (κ) values are found for the negative sentiments, again indicating the least agreements among the tools as is also found in the results of tool-pair agreements (Table 5.4). This can be related to our observations in both Table 5.2 and Table 5.3, where, for all the tools, the accuracy of detecting negative sentiments are found consistently lower compared to non-negative sentiments. Hence, we suspect that all the three tools struggle more or less in accurately detecting especially the negative sentiments in text.

Based on our analyses and observations, we now derive the answer to the second research question (RQ2) as follows:

Ans. to RQ2: *Agreements between the tools in detecting sentiments vary largely across different datasets and sentiments ranging between 49.74% and 94.70%. Much of the disagreements among the tools are attributed to their disagreements in the detection of negative sentiments in text.*

5.5 Threats to Validity

It may be argued that the datasets used in this study are not large enough covering all possible categories technical text relevant to software engineering. However, this study includes all the *publicly available software engineering domain specific* sentiment analysis datasets and tools. However, our study does not include `SentiCR` [111], which is another recently released sentiment analysis tool. We deliberately exclude it since the authors of `SentiCR` declared the scope of this tool limited to code review comments only, and thus it could be unfair to evaluate its performance on datasets including JIRA issue comments or Stack Overflow posts.

Blaz and Becker [163] and Ortu et al. [40] developed tools for sentiment analysis in software engineering related text. The tool and dataset of Blaz and Becker [163] are meant for “Brazilian Portuguese" language and thus not comparable with the datasets and tools used in this study. The tools and datasets of Blaz and Becker [163] and Ortu et al. [40] are not publicly available, which is another reason for excluding them from this study.

5.6 Related Work

We characterize all the SA tool comparison work including ours in four categories: (1) DI-DI: Comparison of domain independent (DI) tools using DI datasets, (2) DI-DS: Comparison of DI tools using SE domain specific (DS) datasets, (3) M-DS: Comparison of mixed (i.e., DI and DS) tools using DS datasets, and (4) DS-DS: Comparison of DS tools using DS datasets.

(1) DI-DI: Abbasi et al. [190] performed a comparison of *domain independent* sentiment analysis (SA) tools by operating them on five different Twitter datasets. Ribeiro et al. [191] conducted a comparison of 24 unsupervised off-the-shelf sentiment analysis methods. Their evaluation was based on labeled datasets including messages posted on social networks, movie and product reviews, as well as opinions and comments in news articles. In an earlier work Gonçalves [192] compared eight sentence-level *domain independent* sentiment analysis methods using a single public DI dataset.

(2) DI-DS: Jongeling et al. [48] compared four *domain independent* SA tools on software engineering dataset and expressed the need for a domain specific SA tool for software engineering text. Islam and Zibran developed `SentiStrength-SE` [110], which is the first software engineering domain specific SA tool (introduced in Section 5.3). In the evaluation, they compared `SentiStrength-SE` against a domain independent tool only. In a later study, Islam and Zibran [57] compared four general purpose SA dictionaries using the JIC dataset introduced in Section 5.2.1.

Recently, Ahmed et al. [111] evaluated seven *domain independent* SA techniques (i.e., *AFINN* [118], *NLTK* [144], *SentiStrength* [45], *TextBlog* [193], *USent* [194], *VADER* [120] and *Vivekn* [195]) using the CRC dataset (Section 5.2.3).

(3) M-DS: Calefato et al. [128], the authors of `Senti4SD` (introduced in Section 5.3), compared their SE domain specific SA tool with *domain specific* `SentiStrength-SE` [110] and *domain independent* `SentiStrength` and `Senti4SD` using the CRC dataset (introduced in Section 5.2.3).

(4) DS-DS: Unlike all the aforementioned work, ours is the first study that compares multiple SE *domain specific* SA tools using multiple publicly available SE *domain specific* datasets. Thus, this work makes a unique contribution to the literature.

5.7 Summary

In this chapter, we have presented the first comparative study of publicly available three software engineering (SE) domain specific sentiment analysis (SA) tools (i.e., `SentiStrength-SE`, `Senti4SD`, and `EmoTxt`) using three SE domain specific datasets.

Our study reveals that the individual tools exhibit their best performance on the dataset they were originally tested at the time of their release. The overall accuracies of the tools tend to decrease when they are operated on a different dataset. The accuracies of the tools largely vary across different datasets and sentimental polarities. Thus, *none* of the tools demonstrates *substantially* superior accuracies across sentiments and datasets. However, `SentiStrength-SE` is found to have consistently exhibited the highest recall and F-score while maintaining competitive precision across all the datasets and sentiments.

From agreement analysis among the tools, we find that the tools' agreements largely vary (between 49.74% and 94.70%) depending on the datasets they are operated on and the sentimental polarities they detect. The tools' agreements remain the lowest in the detection of negative sentiments. Their accuracy also remain lower in the detection of negative sentiments compared to non-negative sentiments. Thus, we suspect that all the tools more or less struggle in accurately detecting negative sentiments in SE text.

We plan to extend this work with new datasets and including an in-depth qualitative analysis to investigate why and in which cases the tools are in disagreements or incorrect in detecting sentiments, especially in the cases of negative ones.

Chapter 6

Detection of Developers' Emotions

In the Chapter 4 and Chapter 5, we described how we developed the first domain specific sentiment analysis tool and measured its performance against its domain specific counterparts. The sentiment analysis tools can detect *valence* (i.e., sentiment) only, and cannot capture arousal or individual emotional states, such as *excitement*, *stress*, *depression*, and *relaxation*. In this chapter, we present the first emotions analysis tool, `DEVA`, which is especially designed for software engineering text and also capable of capturing the aforementioned emotional states through the detection of both arousal and valence

The rest of the chapter is organized as follows. In Section 6.2, we briefly introduce the model of emotions used in this work. In Section 6.3, we introduce our tool, `DEVA`. Our approach for capturing arousal is discussed in Section 6.3.1. The techniques for capturing valence is presented in Section 6.3.2. A set of heuristics included in `DEVA` is described in Section 6.3.4. In Section 6.4, we describe how we empirically evaluate our tool. In Section 6.5, we discuss the limitations of this work. Related work is discussed in Section 6.6. Finally, Section 6.7 concludes chapter.

6.1 Introduction

The techniques for automatic sentiment analysis in text appear to be highly sensitive to domain terms. Thus, the sentiment analysis tools (e.g., `SentiStrength` [45], `NLTK` [46], and `Stanford NLP` [47]), which are designed for general text do not perform well when applied to software engineering text [35, 36, 37, 48, 50, 41, 51, 43] largely due to the variations in meanings of domain-specific technical terms [110]. Hence, recent attempts [163, 110, 111, 128] devise automatic sentiment analysis techniques particularly meant for software engineering text.

All the existing tools are limited in capturing emotions at the necessary depth [50]. Existing approaches are able to detect *valence* (i.e., positivity and negativity of emotional polarities) only and fail to capture *arousal* or specific emotional states such as *excitement, stress, depression,* and *relaxation*. At work, software developers frequently experience these emotions [31], which can be attributed to their work progress. For example, a developer typically feels *relaxed*, if he makes enough progress in his assigned jobs. Otherwise, the developer feels *stressed*. Thus, these emotions need to be identified [121] where the existing approaches fall short [50].

To identify aforementioned emotions with high accuracy we develop a tool and name it `DEVA` (**D**etecting **E**motions in **V**alence **A**rousal Space in Software Engineering Text). The tool includes a lexical approach with a number of heuristics. In empirical evaluations using the aforementioned dataset, `DEVA` demonstrates 82.19% precision, 78.70% recall, and 80.11% F-score. Both the `DEVA` tool and the dataset are made freely available online [196].

6.2 Emotional Model

In this work, we use a simple bi-dimensional model [105, 61] of emotions, which is a variant of the dimensional framework, commonly known as VAD (aka PAD) model [106]. In the bi-dimensional model, as shown in Figure 7.1, the horizontal dimension presents the emotional *polarities* (i.e., positivity, negativity, and neutrality) known as *valence* and the vertical dimension indicates the levels of *reactiveness*, i.e., high and low *arousal*.

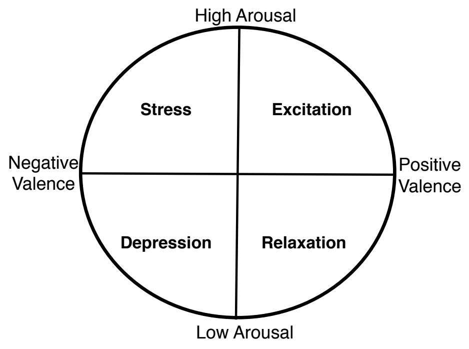


Figure 6.1: Simple bi-dimensional model of emotions

The dimensions are bipolar where the valence dimension ranges from negative to positive and the arousal dimension ranges from low to high. While many emotional states of a person can be determined by combining valence and arousal, we use a set of four major classes of emotional states that include *excitement, stress, depression,* and *relaxation*. For example, positive valence and high arousal, in combination, indicate the emotional state *excitement*. The four emotional states are very distinct, as each state constitutes emotions, which are quite different compared to the emotions of other states [105]. Thus, the model is unequivocal to recognize emotions, simple and easy to understand. This particular emotional model is also used in earlier work [61, 107].

6.3 DEVA

`DEVA` applies a dictionary-based lexical approach particularly designed for operation on software engineering text. For the capturing *both* arousal and valence, the tool uses two separate dictionaries (an arousal dictionary and a valence dictionary) that we develop by exploiting a general-purpose dictionary and two domain dictionaries especially crafted for software engineering text. `DEVA` also includes a preprocessing phase and several heuristics. At the preprocessing phase, `DEVA` identifies and discards source code contents from a given text input using regular expressions similar to what proposed by Bettenburg et al. [151]. The code elements are discarded because they typically are copy-pasted content that do not really carry the writer's emotions [110].

In the following sections, we first describe `DEVA`'s dictionary-based approaches for capturing arousal and valence, and how they are combined to identify different emotional states. Then, we describe the heuristics, which guide the computation of `DEVA` towards high accuracy.

6.3.1 Capturing Arousal

For capturing arousal, we construct a *new* arousal dictionary for `DEVA` by combining the SEA (Software Engineering Arousal) [161] dictionary with the ANEW (Affective Norms for English Words) [160] dictionaries.

The SEA [161] dictionary is specifically developed to detect arousal in text in the software developer ecosystems. The dictionary contains 428 words. Each of the 428 words are assigned an arousal *score* s_a^ω, which is a real number between +1 and +9. In this SEA dictionary, the arousal *level* of a word is interpreted as neutral, if $s_a^\omega = +5$. The arousal level of that word is considered high, if $s_a^\omega > +5$. Otherwise, that word is considered to have low arousal.

The ANEW [160] dictionary is a generic dictionary (i.e., not designed especially for any particular domain), which contains 13,915 words where each word is annotated with arousal, valence, and dominance scores, each also ranging between +1 and +9.

6.3.1.1 Combining the SEA and ANEW dictionaries

At first, all the words (along with their arousal scores) in the ANEW dictionary are included in the new arousal dictionary of `DEVA`. Then, we add any word to the new dictionary if that word is found in the SEA dictionary but not found in the ANEW dictionary. For example, the word 'ASAP' exists in the SEA but not in the ANEW, thus this word along with its arousal score is added to our new arousal dictionary. If a word is found in the both SEA and ANEW dictionaries, then for that word, the arousal score in the SEA dictionary is assigned to the arousal score in our new arousal dictionary. For example, the word 'Anytime' exists in the both dictionaries having the arousal scores 6.5 and 4.6 respectively in the SEA and ANEW dictionaries. Hence, in our new arousal dictionary, the word is assigned an arousal score 6.5. Thus, our newly constructed arousal dictionary includes $14,084$ emotional words.

Table 6.1: Conversion of arousal scores from [+1,+9] to [-5,+5]

Score in [+1,+9]	+1	+2	+3	+4	+5	+6	+7	+8	+9
Score in [-5, +5]	-5	-4	-3	-2	+/- 1	+2	+3	+4	+5

6.3.1.2 Adjusting the ranges of arousal scores

To obtain an arousal scale consistent with valence scale (described later), first, the fractional value of s_a^{ω} is rounded to its nearest integer $\hat{s}_a^{\omega}$. Then, using the conversion scale in Table 6.1, we convert each integer arousal score $\hat{s}_a^{\omega}$ in the range [+1, +9] to $\mathcal{S}_a^{\omega}$ in the integer range [-5, +5]. For example, if the original arousal score of a word rounded to the closest integer is +2, it is converted to -4, according to the mappings shown in Table 6.1. For an arousal score $\mathcal{S}_a^{\omega}$ within the new range of [-5, +5], the arousal *level* $\mathcal{A}_{\omega}$ of a word ω is interpreted using Equation 6.1.

$$\mathcal{A}_{\omega} = \begin{cases} High, & \text{if } \mathcal{S}_a^{\omega} > +1 \\ Low, & \text{if } \mathcal{S}_a^{\omega} < -1 \\ Neutral, & \text{otherwise.} \end{cases} \tag{6.1}$$

This conversion between ranges does not alter the original arousal *levels* of the words.

6.3.1.3 Computing arousal score for *text*

`DEVA` views an input text t as a set of words such as $t = \{\omega_1, \omega_2, \omega_3, ..., \omega_n\}$ where $\omega_1, \omega_2, \omega_3, ..., \omega_n$ are distinct words in t. In computation of the arousal score for the entire text t, `DEVA` retrieves the arousal scores $\mathcal{S}_a^{\omega_1}, \mathcal{S}_a^{\omega_2}, \mathcal{S}_a^{\omega_3}, ..., \mathcal{S}_a^{\omega_n}$ of all the words in t from the arousal dictionary we have constructed. At this particular stage of computation, a word in t is disregarded if it is not found in the arousal dictionary. Then, for t, `DEVA` computes a pair $\langle h_t, \ell_t \rangle$ where,

$$h_t = \max\{\mathcal{S}_a^{\omega_1}, \mathcal{S}_a^{\omega_2}, \mathcal{S}_a^{\omega_3}, ..., \mathcal{S}_a^{\omega_n}\},$$
$$\ell_t = \min\{\mathcal{S}_a^{\omega_1}, \mathcal{S}_a^{\omega_2}, \mathcal{S}_a^{\omega_3}, ..., \mathcal{S}_a^{\omega_n}\}.$$

Finally, `DEVA` determines the overall arousal score $\mathcal{A}_t$ for the *entire* text t using Equation 6.2.

$$\mathcal{A}_t = \begin{cases} h_t, & \text{if } |h_t| \geq |\ell_t| \\ \ell_t, & \text{otherwise.} \end{cases} \tag{6.2}$$

6.3.2 Capturing Valence

To capture valence in text, `DEVA` exploits the only available domain-specific valence dictionary named `SentiStrength-SE`[110], which is especially crafted for software engineering text. This dictionary contains 167 positively and 293 negatively polarized words. Each word ω is assigned a valence score $\mathcal{S}_v^{\omega}$ where $-5 \leq \mathcal{S}_v^{\omega} \leq +5$. Based on the score $\mathcal{S}_v^{\omega}$, the *polarity* (i.e., positivity, negativity, and neutrality) of valence $\mathcal{V}_{\omega}$ of a word is interpreted using Equation 6.3.

$$\mathcal{V}_{\omega} = \begin{cases} Positive, & \text{if } \mathcal{S}_v^{\omega} > +1 \\ Negative, & \text{if } \mathcal{S}_v^{\omega} < -1 \\ Neutral, & \text{otherwise.} \end{cases} \tag{6.3}$$

6.3.2.1 Computing valence score for *text*

The computation of valence scores for a text is similar to the computation of arousal score, except that the valence dictionary is used in place of the arousal dictionary. Thus, for a given text t, `DEVA` computes a pair $\langle \rho_t, \eta_t \rangle$ of integers, where

$$\rho_t = \max\{\mathcal{S}_v^{\omega_1}, \mathcal{S}_v^{\omega_2}, \mathcal{S}_v^{\omega_3}, ..., \mathcal{S}_v^{\omega_n}\},$$
$$\eta_t = \min\{\mathcal{S}_v^{\omega_1}, \mathcal{S}_v^{\omega_2}, \mathcal{S}_v^{\omega_3}, ..., \mathcal{S}_v^{\omega_n}\}.$$

Here, ρ_t and η_t respectively represent the positive and negative valence scores for the text t. Finally, the *overall* valence score $\mathcal{V}_t$ for the text t is computed using Equation 6.4.

$$\mathcal{V}_t = \begin{cases} \rho_t, & \text{if } |\rho_t| \geq |\eta_t| \\ \eta_t, & \text{otherwise.} \end{cases} \tag{6.4}$$

6.3.3 Emotional States from Valence and Arousal

Upon computing the arousal score $\mathcal{A}_t$ and valence score $\mathcal{V}_t$ for a given text t, `DEVA` then maps the emotional scores to individual emotional states based on the bi-dimensional emotional model described in Section 6.2. In particular, the emotional state $\mathcal{E}_t$ (of the author) expressed in the text t is determined using the mapping specified in Equation 6.5.

$$\mathcal{E}_t = \begin{cases} \textit{Excitement}, & \text{if } \mathcal{V}_t \geq +2 \text{ and } \mathcal{A}_t \geq +2 \\ \textit{Stress}, & \text{if } \mathcal{V}_t \leq -2 \text{ and } \mathcal{A}_t \geq +2 \\ \textit{Depression}, & \text{if } \mathcal{V}_t \leq -2 \text{ and } \mathcal{A}_t \leq -2 \\ \textit{Relaxation}, & \text{if } \mathcal{V}_t \geq +2 \text{ and } \mathcal{A}_t \leq -2 \\ Neutral, & \text{if } \mathcal{V}_t = \pm 1. \end{cases} \tag{6.5}$$

In addition to the above mentioned emotional states, a text may express only valence and no arousal, or vice versa. `DEVA` is also able to detect those scenarios in the text.

6.3.4 Heuristics in DEVA

While the underlying dictionaries play the major role in the lexical approach of `DEVA`, the tool also includes a number of heuristics to increase accuracy, as such was also hinted in earlier work [120, 110]. `DEVA` includes all the heuristics implemented in `SentiStrength-SE` [110], which is a recently released tool for the detection of *only valence* in software engineering text. For capturing *arousal* with high accuracy, `DEVA` also includes seven heuristics, which we have devised based on existing studies in psychology and software engineering [39, 197, 198] as well as our experience in the field. These seven heuristics for sensing *arousal* are discussed below with relevant examples excerpted from a dataset [52] composed of JIRA issue comments (described later in Section 6.4.1).

H_1 : The exclamation mark (!) in a text implies high arousal. The exclamation mark (!) in a text is commonly used to indicate high *arousal* of the text writer [199]. For example, in the following comment the commenter expresses *excitement* as the comment contains the word 'happy', which indicates positive *valence*. The three exclamation signs at the end express high *arousal*.

`"Very happy to see it is useful and used !!!."` (Comment ID: 1927)

Thus, by combining positive *valence* (detected using the valence dictionary) and high arousal (detected using this heuristic H_1), `DEVA` correctly identifies (using Equation 6.5) the emotional state *excitement* expressed in the above comment. Without the heuristic H_1 the text would be incorrectly identified to have positive *valence* only.

H_2 : Words with all capital letters indicate high arousal. Words written with all capital letters often indicate high arousal state of the writer [199]. In the following comment, the commenter starts the comment with the word 'sorry' expressing negative valence. All capital letters in the word indicates high arousal state of the commenter.

```
"SORRY Oliver, this is really my fault ...  something like this way
will not happen anymore."
```
(Comment ID: 1802095)

Hence, `DEVA` detects negative valence and high arousal in the comment and identifies that the commenter is under *stress*.

However, in cases, such that API names and code elements are written in all capital letters but they do not express any emotional state of the writer. To distinguish such a scenario, `DEVA` checks the spellings of those words written in all capital letters against an English dictionary using the `Jazzy` [152] tool. A word written in all capital letters, is considered a name of an API or code element, if the word is misspelled. Thus, in the above comment, the word 'SOLR' will be identified as a name and `DEVA` will not interpret it to have expressed any arousal level.

H_3 : Emoticons express emotional states. Emotional icons, aka, emoticons are often used to express different emotions in informal text including software engineering text [198, 105]. For example, in the following comment the writer uses the emoticon ':(' to express *depression*.

`"Oops, I did not run the run-install :("`
(Comment ID: 521081)

Table 6.2: Emoticons expressing different emotions

Emotions	**Code of Emoticons**
Excitement	:*, :?>, :x, :D, :)), 0:), @};- , =P~, :), :">
Stress	:((, X-(, :O, =;, :-/ , (:|, >:P
Depression	:(, :-$, :-&, 8-), :-<, (:|, :-S, I-), :|
Relaxation	B-), :>, :P, ;;), ;), =D>, ;)), :-?, [-o<, /:)

DEVA is capable of identifying and interpreting emotional states expressed in the emoticons used in text. In interpreting the emoticons, DEVA exploits a list of emoticons mapped to the four categories of emotional states (i.e., *excitement, stress, depression/sadness*, and *relaxation*). The mapping, as presented in Table 6.2, was originally proposed by Yang et al. [197].

H_4 : Interjections can indicate emotional states. The interjections are special parts of speech (POS), which are meant for expressing emotional states [200]. For example, even if the above comment (ID: 521081) did not include the emoticon ':(', it would still express *depression* through the interjection 'oops', which DEVA would capture correctly using a list of interjections mapped to their meanings [201] as presented in Table 6.3.

Table 6.3: Interjections expressing different emotions

Emotions	**Interjections**
Excitement	'Gee', 'Hurray', 'Ooh', 'Oooh', 'Wee', 'Wow', 'Yah', 'Yeah', 'Yeeeeaah', 'Yee-haw'
Stress	'Aah!', 'Aaah', 'Aaaahh', 'Argh', 'Augh', 'Bah', 'Boo', 'Boo!', 'Booh', 'Eek', 'Eep', 'Grr', 'Yikes'
Depression	'Duh', 'Doh', 'Eww', 'Gah', 'Humph', 'Harumph', 'Oops', 'Oww', 'Ouch', 'Sheesh', 'Jeez', 'Yick'
Relaxation	'Ahh', 'Phew'

The list of interjections in Table 6.3, includes only those interjections, whose meanings are unambiguous and relevant to the emotional sates considered in this work. For example, the interjection 'Yahoo' is excluded since it sometimes expresses the name of a company.

H_5 : Temporal terms indicate high arousal. In psychology and management, it is generally accepted that because of time pressure a worker shows increased alertness or readiness i.e., high arousal [39]. This also applies to the software engineering field [161, 202, 203]. Thus, high arousal is expressed through temporal terms (e.g., asap, soon) in text. For example, in the following comment the commenter expresses high arousal by using the temporal term 'asap'.

```
"Sorry, I will fix these error asap."
```
(Comment ID: 1600615)

To capture such arousal states expressed through temporal terms, `DEVA` maintains a list of 12 temporal terms (Table 6.4) commonly used for referring to timelines, deadlines, or such.

Table 6.4: Temporal terms included in the `DEVA` dictionary

'Soon', 'Sooner', 'ASAP', 'EOD', 'EOB', 'Today', 'Tomorrow', 'Tonight', "No later", "At earliest", "Tight schedule"

H_6 : Task completion leads to low arousal. Typically, completion of a task makes one relaxed and at the state of low *arousal*. For example, the following comment indicating completion of a task also expresses the relaxation of the commenter.

`"The two approaches seem complimentary to me. I'm happy to see this committed. Does anyone object?"` (Comment ID: 1667040)

The word 'happy' in the above comment indicate positive valence. But, the arousal state could be missed out if the word 'committed' is not considered to have expressed low arousal. `DEVA` takes into account both the words 'happy' and 'committed' and corrected identifies both positive valence and low arousal jointly mapped to *relaxation*. For capturing the task completion scenarios in software engineering, `DEVA` uses a collection of domain-specific words and phrases listed in Table 6.5.

Table 6.5: Task completion indication terms in `DEVA`

'Fixed', 'Resolved', 'Solved', 'Done', "Patch looks good", "Working fine", "Working good", "Working properly", "Pushed in branch", "Pushed in trunk", 'Committed'.

H_7 : Negations reverse arousal state. Generally, a negation (e.g., no, not) is meant for reversing or weakening the meaning of the word it qualifies.

`DEVA` weakens the arousal level associated with a word when the word is found negated in text. Thus, *high* arousal level associated with a negated word is weakened to *low* arousal and the *low* arousal of a word is *neutralized*. For example, in the comment below the high arousal word 'worry' (it is also a negative *valence* word) is negated by 'not' and indicates low arousal.

`"Lets not worry about this now"` (Comment ID: 53698)

Again, in the following comment, the word 'good' is associated with a low arousal level in `DEVA`'s arousal dictionary. Identifying the negation of the word with 'not', `DEVA` neutralizes the low arousal.

`"Agreed, its not good. Improved in 1.1."`
(Comment ID: 2263164)

6.4 Evaluation

The accuracy of emotion detection of DEVA is measured in terms of *precision* ($\wp$), *recall* ($\Re$), and *F-score* ($\Finv$) separately computed for each of the target emotional states as described in Section 6.2 and formalized in Equation 6.5. Given a set $\mathcal{I}$ of texts, *precision* $\wp$, *recall* $\Re$, and *F-score* ($\Finv$) for a particular emotional state e is calculated as follows:

$$\wp = \frac{|\ \mathcal{I}_e \cap \mathcal{I}'_e\ |}{|\ \mathcal{I}'_e\ |}, \quad \Re = \frac{|\ \mathcal{I}_e \cap \mathcal{I}'_e\ |}{|\ \mathcal{I}_e\ |}, \quad \Finv = \frac{2 \times \wp \times \Re}{\wp + \Re},$$

where $e \in \{excitement, stress, depression, relaxation, neutral\}$, $\mathcal{I}_e$ represents the set of texts expressing the emotional state e, and $\mathcal{I}'_e$ denotes the set of texts for which DEVA correctly captures the emotional state e.

Recall that DEVA is the first tool capable of automatic detection of the aforementioned emotional states in software engineering text, and no dataset is available for empirical evaluation of our tool. Hence, we first create a ground-truth dataset and compute the aforementioned metrics against that. Then we compare DEVA with a baseline approach we also implement. Finally, we compare our tool with a similar (but not identical) tool, `TensiStrength` [122].

6.4.1 Creation of Ground-Truth Dataset

The considered dataset [52] consists of two million JIRA issue comments over more than 1,000 projects. JIRA[1] is a commercial tool widely used by software developers for describing, tracking, and managing user-stories, bug-reports, feature requests, and other development issues. This dataset has also been used in many studies [110, 39, 185, 63, 40] on the social and emotional aspects of software engineering.

6.4.1.1 Construction of a manageable subset

We want to create a dataset by manually annotating the issue comments with their expressed emotional states. Manual annotation of two million issue comments could be a mammoth task. Hence, to minimize efforts, we create a subset of 2,000 issue comments for manual annotation using some criteria as described below.

The majority of issue comments in the above mentioned dataset are emotionally neutral [52]. Thus, a random selection is likely to include more neutral comments than those with other emotions. To avoid such a possibility, we first use a keyword-based searching method to collect from the original dataset a subset $\mathcal{G}_k$ of 50 thousand comments which are likely to contain *valence* and *arousal*. We use 68 unigram keywords (listed elsewhere [105]) and their 136 synonyms detected using Word-

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

Table 6.6: Inter-rater disagreements in categories of emotions

	Inter-rater Disagreements			**# of Issue**
Emotions	**A, B**	**B, C**	**C, A**	**Comments**
Excitement	06.57%	07.79%	07.54%	411
Stress	06.75%	17.46%	14.68%	252
Depression	06.92%	11.07%	07.61%	289
Relaxation	05.72%	09.69%	05.72%	227
Neutral	07.30%	05.19%	05.68%	616
Total number of issue comments:				1,795

Net [204]. The synonyms of a keyword include every synonym of all variations of the keyword with respect to POS. Such a keyword-based searching method is also used in another study [105] for a similar purpose.

Again, from the original dataset, we randomly select another subset $\mathcal{G}_r$ of 100 thousand issue comments. Then we create a set $\mathcal{G}_u$ such that $\mathcal{G}_u = \mathcal{G}_k \cup \mathcal{G}_r$. Then from $\mathcal{G}_u$, we filter out those comments, which have more than 100 letters resulting in another set $\mathcal{G}_{\hat{u}}$ consisting of 110 thousand comments. From the set $\mathcal{G}_{\hat{u}}$, we randomly select 2,000 comments for manual annotation by human raters.

6.4.1.2 Manual annotation by human raters

We employ three human raters (enumerated as A, B, C) for manually annotating the 2,000 issue comments with the emotions (i.e., *excitement, stress, depression*, *relaxation*, or *neutral*) they perceive in them. Each of these three human raters are graduate students in computer science having one to five years experience in software development in collaborative environments. Each of the human raters separately annotate each of the 2,000 issue comments.

We consider a comment conveying the emotional state *e*, if two of the three raters identify the same emotion in it. Total 205 issue comments are discarded since the human raters do not agree on the emotions they perceive in those comments. Thus, our ground-truth dataset ends up containing 1,795 issue comments. The number of issue comments expressing each of the emotional states are presented in the rightmost column of Table 6.6. This table also presents the emotion-wise percentage of cases where raters disagree. We also measure the degree of inter-raters agreement in terms of *Fleiss-κ* [150] value. The obtained *Fleiss-κ* value 0.728 signifies substantial agreement among the independent raters.

6.4.2 Measurement of Accuracy

We invoke DEVA to detect the emotional states in each of the 1,795 issue comments in our human-annotated ground-truth dataset. Then, for each of the issue comments, we compare DEVA's detected emotion with the human annotated emotion (i.e., ground-truth). We separately measure precision ($\wp$), recall ($\Re$), and F-score ($\Finv$) for DEVA's detection of each of the emotional states, which are presented in the third column (from the left) of Table 6.7. As presented at the bottom three rows in the same column of the table, across all the emotional states, on average, DEVA achieves 82.19% precision, 78.70% recall, and 80.11% F-score.

Table 6.7: Comparison beween DEVA and Baseline

Emotions	**Metrics**	**DEVA**	**Baseline**
Excitement	$\wp$	87.58	77.16
	$\Re$	88.86	23.72
	$\Finv$	88.22	36.29
Stress	$\wp$	72.29	48.48
	$\Re$	66.53	12.74
	$\Finv$	69.29	20.18
Depression	$\wp$	78.01	33.77
	$\Re$	76.12	61.59
	$\Finv$	77.05	43.62
Relaxation	$\wp$	85.63	19.63
	$\Re$	65.63	66.76
	$\Finv$	74.31	30.33
Neutral	$\wp$	87.44	72.45
	$\Re$	96.37	31.63
	$\Finv$	91.69	44.03
Average	$\wp$	82.19	50.30
	$\Re$	78.70	39.27
	$\Finv$	80.11	34.87

6.4.3 Comparison with a Baseline

DEVA is the first tool especially designed for *software engineering text* to detect the emotional states in the bi-directional emotion model encompassing *both* valence and arousal. There exists no such other tool for direct comparison with DEVA. Hence, we implement a baseline approach based on the work of Mäntylä et al. [39] who used the ANEW dictionary to only *study* valence and arousal in software engineering text.

The baseline tool that we implement also exploits the ANEW dictionary. Thus, the baseline tool differs from DEVA in two ways. First, the baseline tool uses the regular ANEW dictionary while DEVA exploits a valence dictionary and an arousal dictionary especially designed for software engineering text. Second, DEVA applies a number of heuristics which are not included in the baseline tool. We want to verify if the crafted dictionaries and heuristics actually contribute to higher accuracy in the detection of emotional states.

Hypothesis: Upon operating DEVA and the baseline tool on the same software engineering dataset, DEVA must outperform the baseline, if the domain-specific dictionaries and heuristics included in it actually contribute to higher accuracies in the detection of emotional states in software engineering text.

We invoke the baseline tool to detect the emotional states in each of the issue comments in our ground-truth dataset. Then, we compute the precision ($\wp$), recall ($\Re$), and F-score ($\dashv$) for its detection of each emotional states (i.e., excitement, stress, depression, relaxation, and neutral) as shown in the rightmost column of Table 6.7. The overall average precision, recall, and F-score across all the emotional states are presented in the bottom three rows of the same column.

As we compare the accuracies of DEVA and the baseline approach in Table 6.7, our DEVA is found to have outperformed the baseline in all cases by a large margin except for the recall of relaxation where DEVA falls short by only 01.13%. In all cases, DEVA maintains a substantially higher F-score compared to the baseline. In other words, DEVA maintains a balance between precision and recall for each emotional state resulting in higher F-score for all cases. Overall, on average, across all the emotions, DEVA clearly outperforms the baseline.

Thus, the results of comparison imply that our hypothesis holds true, which means the domain-specific dictionaries and heuristics included in DEVA actually contribute to its superior performance.

6.4.4 Comparison with TensiStrength

Recently, TensiStrength [122] is released, which we find somewhat similar to our DEVA because both the tools are capable of detecting *stress* and *relaxation* in text. However, DEVA and TensiStrength are more different than they are similar. First, unlike DEVA, the TensiStrength tool is not especially designed for software engineering text. Second, TensiStrength cannot detect *excitement* and *depression*, which DEVA detects. Nevertheless, we compare TensiStrength's accuracies against those of DEVA in the detection of *stress* and *relaxation* only since these emotional states form a subset of the emotional states DEVA detects.

For a given text t, TensiStrength computes a pair $\langle \pi_t, \varsigma_t \rangle$ of integers, where $+1 \leq \pi_t \leq +5$ and $-5 \leq \varsigma_t \leq -1$. Here, π_t and ς_t respectively represent the *relaxation* and *stress* scores for the given text t. A given text t is considered expressing *relaxation* if $\pi_t > +1$. Similarly, a text is held conveying *stress* when $\varsigma_t < -1$. Besides, a text is considered neutral when the scores for the text appear to be $\langle 1, -1 \rangle$.

We execute TensiStrength on the ground-truth dataset. Then, we separately measure the precision ($\wp$), recall ($\Re$), and *F-score* ($\dashv$) for TensiStrength's detection of each of the three target emotional states (i.e., relaxation, stress, and neutral). Table 6.8 shows the precision, recall, and F-score of both the tools DEVA and TensiStrength in the detection of *stress*, *relaxation* and *neutral*

Table 6.8: Comparison between `DEVA` and `TensiStrength`

Emotions	**Metrics**	**DEVA**	**TensiStrength**
Stress	$\wp$	**72.29**	35.70
	$\Re$	66.53	**92.03**
	$\Finv$	**69.29**	51.44
Relaxation	$\wp$	**85.63**	20.58
	$\Re$	**65.63**	62.11
	$\Finv$	**74.31**	30.92
Neutral	$\wp$	**87.44**	82.31
	$\Re$	**96.37**	79.73
	$\Finv$	**91.69**	81.00
Average	$\wp$	**81.79**	46.20
	$\Re$	76.18	**77.96**
	$\Finv$	**78.43**	54.45

comments. The overall average precision, recall, and F-score across the target emotional states are presented in the bottom three rows of the table.

As seen in Table 6.8, `DEVA` consistently achieves higher precision and F-score in the detection of all the emotional states. The recall of `DEVA` is also higher in all cases except for recall of *stress*, which affects the comparative overall recall of the tools. Still, `DEVA` maintains higher overall average F-score.

`TensiStrength` cannot differentiate between *depression* and *stress*. It cannot distinguish between *excitement* and *relaxation* either. These shortcomings are among the reasons for the tool's lower precision in the detection of *stress* and *relaxation*. For example, in the following comment, the commenter conveys *excitement*, but due to presence of the positive emotional word 'good', `TensiStrength` incorrectly determines the comment to have expressed *relaxation*.

```
"Good catch !  Will fix it asap."
```
(Comment ID: 1348887)

6.5 Threats and Limitations

From the empirical evaluations, `DEVA` is found superior to both the baseline approach and `TensiStrength`. Still, its accuracy is not 100% due to its shortcomings. Although `DEVA` captures negations very well, it still falls short in handling *complex structures* of negations. In the detection of subtle expressions of emotions in text, even the human raters are often in disagreements, and `DEVA` also falls short in capturing them. The tool cannot distinguish irony and sarcasm in text, and fails to correctly identify emotions in such text. Capturing subtle emotional expressions, irony, and sarcasm in text is already recognized as a challenging problem in the area of Natural Language Processing (NLP).

The heuristics and domain-specific dictionaries included in DEVA contribute in correct identification of emotional states as verified in Section 7.4.4. However, in some cases, the heuristics may mislead the tool, although such cases are relatively rare compared to the common situations. The lists of task completion terms, temporal terms, interjections, and emoticons, included in DEVA, might not be complete to cover all possible scenarios. Similarly, the valence and arousal dictionaries in DEVA might also miss relevant emotional terms. One might question, instead of using the lexical approach for building DEVA's domain-specific dictionaries, if we could adopt any better approach, which could possibly minimize these limitations. However, a recent study [57] reports that, "lexicon-based approaches for dictionary creation work better for sentiment analysis in software engineering text."

One might argue that in construction of DEVA's arousal dictionary, the range conversion of arousal scores from [+1, +9] to [-5, +5] might have altered the original arousal levels of some words. We have considered this possibility and carefully designed the conversion scheme to minimize such possibilities. A random sanity check after the range conversion indicates absence of any such occurrence. The regular expressions used in the preprocessing phase of DEVA for filtering out source code elements in text might not be able to discard all code elements. However, studies show that light-weight regular expressions perform better than other heavy-weight approaches (e.g., machine learning, island grammar) for this purpose [205].

Our ground-truth dataset manually annotated by three human raters are subject to human bias, experience, and understanding of the field. However, the human raters being computer science graduate students and having software development experience in collaborative environments limit this threat.

6.6 Related work

A comprehensive list of the tools and techniques developed and used to detect emotions can be found elsewhere [206, 121, 207]. To maintain relevance, we limit our discussion to only those tools and techniques that are attempted for *software engineering text*.

Earlier research involving sentiment analysis in software engineering text used three tools/toolkits, SentiStrength [45], Stanford NLP [146], and NLTK [46], while SentiStrength is used the most frequently [110]. All of the aforementioned three tools are developed and trained to operate on non-technical text and they do not perform well enough when operated in a technical domain such as software engineering. Domain-specific (e.g., software engineering) technical uses of inherently emotional words seriously mislead the sentiment analyses of those tools [48, 50, 41, 43] and limit their applicability in software engineering area.

Blaz and Becker [163] proposed three almost equally performing lexical methods, a Dictionary Method (DM), a Template Method (TM), and a Hybrid Method (HM) for sentiment analysis in

"Brazilian Portuguese" text in IT (Information Technology) job submission tickets. Although their techniques might be suitable for *formally structured* text, those may not perform well in dealing with *informal* text frequently used in software engineering artifacts such as commit comments [110]. `SentiStrength-SE` [110], `Senti4SD` [128] and `SentiCR` [111] are three recent tools especially designed to deal with software engineering text. However, all the aforementioned tools and techniques are meant for detecting *valence only* and cannot capture arousal or other emotional states at a deeper level.

To detect emotions in more fine-grained levels, Murgia et al. [185] constructed a machine learning classifier specifically trained to identify six emotions *joy, love, surprise, anger, sad,* and *fear* in issue comments. Similar to their approach, Calefato et al. [56] also developed a toolkit to detect those six emotions. However, neither of these techniques are capable of detecting the emotional states *excitement, stress, depression,* and *relaxation* as captured in the well-established bi-directional emotional model encompassing both *valence* and *arousal* dimensions.

`TensiStrength` [122] is a recently released tool, which we have compared with our `DEVA`. As mentioned before, `TensiStrength` can detect *stress* and *relaxation* from text, but cannot capture *excitement* or *depression*, while `DEVA` is capable of detecting all of them. Unlike our `DEVA`, `TensiStrength` is not especially designed for any particular domain, and thus performs poorly for software engineering text as such is also found in our comparison with `DEVA`. Mäntylä et al. [39] studied both valence and arousal in software engineering text. For detection valence and arousal they also used a lexical approach, which is not especially designed for software engineering text. Their approach relies on the ANEW (Affective Norms for English Words) dictionary only, whereas `DEVA` uses two separate valence and arousal dictionaries especially crafted for software engineering text. Although their approach was never realized in a reusable tool, it inspired us in the implementation of the baseline tool that we have compared with `DEVA`.

6.7 Summary

In this chapter, we have presented `DEVA`, a tool for automated sentiment analysis in text. `DEVA` is unique from existing tools in two aspects. First, `DEVA` is especially crafted for software engineering text. Second, `DEVA` is capable of detecting both valence and arousal in text and mapping them for capturing individual emotional states (e.g., *excitement, stress, depression, relaxation* and *neutrality*) conforming to a well-established bi-directional emotional model. *None* of the existing sentiment analysis tools have *both* the aforementioned capabilities/properties. `DEVA` applies a lexical approach with an arousal dictionary and a valence dictionary, both crafted for software engineering text. In addition, `DEVA` includes a set of heuristics, which help the tool to maintain high accuracy.

For empirical evaluation of `DEVA`, we have constructed a ground-truth dataset consisting of 1,795 JIRA issue comments, each of which are manually annotated by three human raters. This dataset is also a significant contribution to the community. From a quantitative evaluation using this dataset, `DEVA` is found to have achieved 82.19% precision and 78.70% recall. We have also implemented a baseline approach and compared against `DEVA`. A recently released similar (but not identical) tool `TensiStrength` is also compared with our `DEVA`. From the comparisons, `DEVA` is found substantially superior to both the baseline and `TensiStrength`.

The current release of `DEVA` and our ground-truth dataset are freely available [196] for public use. We are aware of the existing limitations of our tool, which we have also discussed in this chapter. Addressing all these limitations is within our future plan. In the future releases of `DEVA`, we will keep enriching the underlying dictionaries and enhancing the heuristics for further improving the tool's accuracy. Using `DEVA` and its future releases, we will conduct large scale studies of emotional variations and their impacts in software engineering. Moreover, we have plan to extend `DEVA` for *aspect-oriented* [208, 209] emotion analysis in software engineering text.

Chapter 7

Machine Learning Based Detection of Developers' Emotions

Our tools `SentiStrength-SE` and `DEVA` (described in Chapter 4 and Chapter 6, respectively) are developed by combining domain specific rules and dictionaries. However, a machine learning technique can overcome the default limitations of a tool (e.g., `DEVA`) developed using such rules and dictionaries. In this chapter, We describe the development procedure of `MarValous`, the first *Machine Learning* based tool for improved detection of *excitement*, *stress*, *depression*, and *relaxation* emotional states in software engineering text.

The remainder of the chapter is organized as follows. In Section 7.1, we introduce our motivation and context of the work. In Section 7.2, we briefly introduce the two-dimensional emotional model used in this work. In Section 7.3, we introduce our tool, `MarValous` and describe our machine learning approach for the detection of individual emotional states. In Section 7.4, we describe how we empirically evaluate our tool. In Section 7.5, we discuss the limitations and threats to the validity of this work. Section 7.6 includes a discussion of related work. Finally, Section 7.7 concludes the chapter.

7.1 Introduction

Recently, a few SE domain-specific tools [163, 110, 111, 128] are developed for detecting *sentimental polarities* (e.g., positivity, negativity) only. Those tools are limited in capturing emotional states at the necessary levels such as in capturing *excitation, stress, depression,* and *relaxation* while it is important to capture these emotional states [129, 50]. Islam and Zibran recently addressed this limitation and developed `DEVA` [129], which, till date, is the only tool available for detecting those four emotional states in SE texts. However, `DEVA` is lexicon-based and such a technique is limited by the quality and sizes of the underlying dictionaries in use [187, 210]. Arguably, a dictionary-based approach can fail to capture the organization of a text that contributes information relevant to the emotion of the text writer [211].

In this chapter, we present `MarValous` (**Ma**chine Lea**r**ning Based Emotion Detector in **Val**ence-Ar**ous**al Space), a tool that we have developed for automatic detection of individual emotional states expressed in SE text. In particular, this work makes the following three contributions:

Our MarValous tool exploits supervised ML techniques. It includes nine text preprocessing steps and seven feature extraction modules. In empirical evaluations using the aforementioned unified dataset, MarValous demonstrates 83.37% precision, 79.33% recall, and 80.90% F-score. Both the MarValous tool and the dataset are made publicly available [196].

7.2 Emotional Model

We use a two-dimensional model [105, 61] of emotions to classify texts in software engineering. Figure 7.1 presents the most widely used emotion classification model in the two-dimensional approach [212] proposed by Russell and Mehrabian [106]. As shown in Figure 7.1, each dimension is bipolar where the valence dimension ranges from negative valence (i.e., pleasant) to positive valence (i.e., unpleasant) and the arousal dimension ranges from low to high. Total 28 emotional states of a person can be determined by combining different levels of valence and arousal in each of the four quadrants marked as Q1, Q2, Q3, and Q4 in Figure 7.1.

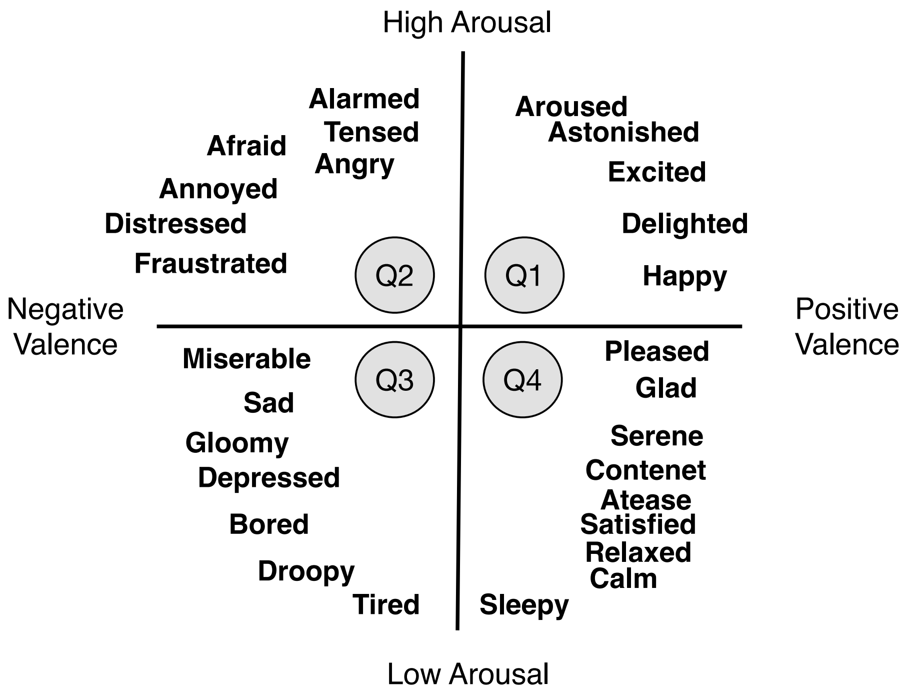


Figure 7.1: Two-dimensional emotion classification model.
0

Several studies [129, 105, 213] use simplified versions of the aforementioned two-dimensional model of Russel and Mehrabian [106], where each quadrant is represented by a unique emotional state. For example, the quadrants Q1, Q2, Q3 and Q4 are represented by emotions *excitation, stress, depression,* and *relaxation* respectively, in the work of Islam and Zibran [129]. The four classes of emotions are very distinct, as each state constitutes emotions, which are quite different compared to the emotions of other states. Moreover, the model is simple and easy to perceive. Therefore, the simplified model of Islam and Zibran [129] is also adopted in our work.

7.3 Marvalous

We develop `MarValous` in Python and use *scikit-learn* [214] for supervised learning algorithms. For improved classification performances, `MarValous` consists of two major modules: (i) data preprocessing and (ii) feature selection, which are described in the following subsections.

7.3.1 Preprocessing

In the preprocessing phase, we sanitize the input text to get rid of probable noises, which otherwise could mislead classification.

URL and code snippet removal: Natural language texts (e.g., commit and issue comments) generated during software development may often include noisy texts such as URL references and code snippets, which do not convey any emotion of writers of those texts. However, those URLs and code snippets may contain emotional words that can easily mislead emotion detection approaches/-tools [109]. Moreover, keeping those noises in texts will increase the size of features' vector for a ML classifier. We use a simple regular expression technique to identify and remove all URLs and code snippets from our dataset. Such simple regular expression technique is found to be effective [151] and also used for similar purpose in other studies [109, 111] too.

Removal of numeric expressions: Similar to URLs and code snippets, any numbers in texts do not indicate any emotion of the writers and increase size of features' vecrtor. Hence using a regular expression, we identify and remove all numbers from our dataset.

Slang removal: Due to informal nature of communications among developers, they frequently use slangs in their writings. For example, in the comment, "`Thanx a lot!  Could you tell me, how can I download ...  OM 2.2?  :`," the writer uses the slang 'Thanx' instead of the English word 'Thanks'. We replace such slangs in texts with formal English words to reduce features' vector size.

Stop word removal: Removing stop-words (such as articles, prepositions, and conjunctions) to reduce number of features is a common practice in ML based techniques. While predicting emotions in texts, such removal of stop-words is highly expected as those stop-words do not play any significant roles to express emotions.

Although popular natural language processing tools (e.g., `Stanford CoreNLP` [146] and `NLTK` [46]) provide lists of stop-words, we use a customized stop-word list presented in Table 7.1. As the popular tools' provided stop-words list includes personal pronouns (e.g., 'he', 'she', and 'my'), temporal terms (e.g., now), booster words (e.g., very) and few others terms that are used for negations (e.g., 'no', 'not') and asking questions (e.g., 'why', 'what'), which play important roles

Table 7.1: A customized stop-words list

‘it,’ ‘itself,’ ‘this,’ ‘that,’ ‘these,’ ‘those,’ ‘is,’ ‘are,’ ‘was,’ ‘were,’ ‘be,’ ‘been,’ ‘being,’ ‘have,’ ‘has,’ ‘had,’ ‘having,’ ‘do,’ ‘does,’ ‘did,’ ‘doing,’ ‘a,’ ‘an,’ ‘the,’ ‘and,’ ‘if,’ ‘or,’ ‘as,’ ‘until,’ ‘while,’ ‘of,’ ‘at,’ ‘by,’ ‘for,’ ‘between,’ ‘into,’ ‘through,’ ‘during,’ ‘to,’ ‘from,’ ‘in,’ ‘out,’ ‘on,’ ‘off,’ ‘then,’ ‘once,’ ‘here,’ ‘there,’ ‘all,’ ‘any,’ ‘both,’ ‘each,’ ‘few,’ ‘more,’ ‘other,’ ‘some,’ ‘such,’ ‘than,’ ‘too,’ ‘s,’ ‘t,’ ‘can,’ ‘will,’ ‘don,’ ‘should’

in expressing emotions [110, 129, 215]. Thus, those types of terms are removed from stop-words collection to prepare our customized list.

Name replacement: Developers typically mention their colleagues’ names in texts immediately after salutation words such as ‘Dear’, ‘Hi’, ‘Hello’, ‘Hellow’ or after the character ‘@’ [110], which do not convey any emotion rather increase size of features’ vector. Hence, all words that start after the words ‘Dear’, ‘Hi’, ‘Hello’, ‘Hellow’ and the symbol ‘@’ are replaced by the single word ‘UserName’.

Software-specific named entity replacement. Similar to colleagues’ names, software-specific named entities do not express any emotion in texts and increase size of features vector. Hence, we identify the named entities and replace those using the keyword ‘NamedEntity’. To identify those named entities, we use the gazetteer [216] prepared for software engineering domain. The gazetteer includes total 400,147 entries divided into five categories, which are presented in Table 7.2. However, all the entries in the API category are detected and removed by the logic of detection of code snippet.

Table 7.2: Categories of Software-specific Name Entities

Named-Entity Category	**# of Entries**
Programming language (e.g., Java, C)	419
Platform (e.g., x86, AMD64)	175
API (e.g., Java ArrayList, toString())	396,968
Tool-library-framework (e.g., JProfiler, Firebug)	2,196
Software standard (e.g., HTTP, FTP)	389

Dealing with negations: Generally, a negation word (e.g., ‘no’, ‘never’) is used for reversing or weakening the meaning of the word it qualifies [110]. Since ML based classifiers operate on unigrams and bi-grams representations of sentences, those classifiers often fail to identify negated opinions [111]. To overcome that problem, an earlier ML based sentiment analysis tools [125] adopted a simple approach by prepending 'not' with the succeeding words that are found after a negation word.

However, negations only affect verbs, adjectives, and adverbs but do not alter nouns, determiners, articles, and particles [111]. Thus, we modify verbs, adjectives, and adverbs (instead of all the words) by prepending 'not_', which are found within the scope of a negation word in a sentence. To determine scope of negations, we use *chunking* or *shallow parsing* [111] to divide a text into syntactically correlated parts of words. The work of Ahmed et al. [111] also follows the same procedure.

Word tokenizing and stemming: Among available popular tokenizer tools e.g., `NLTK`, `Stanford CoreNLP`, `SyntaxNet` [217] and `spaCy` [218], we use NLTK as it shows the best performance [219] in software engineering domain to tokenize words in texts. After tokenization, we apply word stemming [220] to convert each word to its root. We apply Snowball Stemmer [220] as it is used in another software engineering study [111].

Expansion of contractions: Contractions are shortened forms of a group of words, which are commonly used in informal written communications. When a contraction is written in English, the omitted letters are replaced by an apostrophe. Some frequently used contractions and their expanded forms include: aren't → are not, and I'm → I am and won't → will not. Writing the same thing in two different ways (i.e., using or not using contraction) increases size of features vector.

Thus, such expansion of contractions reduces the number of unique lexicons (i.e., feature vectors), which, in turns, helps to improve ML based classifiers' performances. We expand total 124 frequently used contractions [111] if they are found in texts.

7.3.2 Feature Selection

For machine learning, we identify a set of features in text, which we describe below along with the rationale why they can be useful in emotion classification.

n-gram: An n-gram is a contiguous sequence of *n* items from a given piece of text. The items can be phonemes, syllables, letters, words or base pairs and *n* can be any natural number. In our work, an item refers to a *word* in a given text and *n* ranges from one to two i.e., we use *unigram* (where $n = 1$) and *bigram* (where $n = 2$) as features.

For example, if a given text consists of the words w_1 w_2 w_3, then unigram feature will contain each of the words i.e., w_1, w_2 and w_3 and bigram feature will contain pair of words i.e., w_1 w_2 and w_2 w_3. We compute TF-IDF (Term Frequency - Inverse Document Frequency) [221] for each of the unigram and bigram features. The n-gram method is language independent and works well in the case of noisy-texts [222]. Thus, it suits very well due to informal and noisy natures of software engineering texts.

Emoticons: In informal written communications emoticons are frequently used to express different emotions of writers. Hence, emoticons have been commonly used to classify emotions in several studies [129, 111, 105, 128]. We use total 38 emoticons in this work and categorize those in the four quadrants according to their emotions as presented in Table 7.3 (see the first and third columns). This categorization of emoticons is used in other studies [129, 198] too. We use a binary feature (e.g., hasEmoticon) that keeps record of existence of emoticons in a given text.

Moreover, to reduce features' vector size, we replace 38 emoticons in texts with four keywords according to their emotions. For example, if we find an emoticon, which expresses emotion *excitation*, then that emoticon is replaced with the keyword 'Excited'. All such four keywords are mentioned in the second column of Table 7.3 with respect to their emotions.

Table 7.3: Emoticons and interjections expressing different emotions

Emotion	**Keyword**	**Emoticon**	**Interjection**
Excitation	Excited	:*, :?>, :x, :D, :)), 0:), @};- , =P~, :), :">	'Gee', 'Hurray', 'Ooh', 'Oooh', 'Wee', 'Wow', 'Yah', 'Yeah', 'Yeehaw'
Stress	Stressed	:((, X-(, :O, =;, :-/ , (:|, >:P	'Aah!', 'Aaah', 'Argh', 'Augh', 'Bah', 'Boo', 'Boo!', 'Booh', 'Eek', 'Eep', 'Grr', 'Yikes'
Depression	Depressed	:(, :-$, :-&, 8-), :-<, (:|, :-S, I-), :|	'Duh', 'Doh', 'Eww', 'Gah', 'Humph', 'Harumph', 'Oops', 'Oww', 'Ouch', 'Sheesh', 'Jeez', 'Yick'
Relaxation	Relaxed	B-), :>, :P, ;;), ;), =D, ;)), :-?, [-o<, /:)	'Ahh', 'Phew'

Interjections: The interjections are special parts of speech (POS), which are also frequently used to express emotional states [129]. As presented in the fourth column of Table 7.3, we use total 37 interjections that are categorized into four emotions according to their meanings in an earlier work [129]. Similar to emoticons, we use another binary feature (e.g., hasInterjection) that identifies presence of interjections in a given text. Again, to reduce features' vector size, we replace the interjections with four keywords presented in the second column of Table 7.3 (similar to replacement process of emoticons) according to their emotions.

Exclamation marks: Writers often use exclamation mark when they want to express their intense feelings [105, 128]. For example, the comment, "`I will fix it immediately!`," expresses a high level of *stress* of the writer. On the other hand, the comment, "`Yonik, Thanks and Congratulations!`," indicates a high level of *excitation*. In both cases, the writers of the comments use exclamation marks to put more emphasize on their feelings. Thus, we use a binary feature (e.g., hasExclamation) that captures presence of exclamation marks in texts.

Uppercase words: To put higher emphasize on emotional expressions, writers sometimes write few words using all uppercase/capital letters [129, 128] (e.g., GOOD, AWESOME, and BAD). For example, in the comment, "`SORRY Oliver, this is really my fault`," the writer expresses a higher level of sadness by writing the word 'Sorry' using all capital letters. We identify if any word written in all capital letters in a piece of text and record that information to use as a binary feature (e.g., hasAllCapitalLettersWord). In many cases, API names and code elements are written in all capital letters, which are discarded at preprocessing phase (see subsection 7.3.1).

Elongated words: Similar to uppercase word, writers use elongated words (e.g., Goood, Hurraaaay) to express intense emotions in informal written communications [128]. We identify existing of such elongated words in texts and record that information to use as a binary feature (e.g., hasElongatedWord). Similar to contractions, elongated words also adversely contribute to increase the size of features vector. For example, the elongated word 'Goood' will be considered as a unique word/feature, although it is a deviated form of the English word 'Good'. To reduce size of features vector, we correct the spellings of such identified elongated words found in texts.

Use of $+1$ and -1 in sentences: It is a common practice of developers to put $+1$ and -1 in comments while discussing technical issues among them in Stack Overflow and JIRA. When a developer likes or agrees on any issue with his colleague(s), then he puts $+1$ while commenting on that issue [223]. For example, in the comment "`+1, the new patch looks good`," the writer uses $+1$ (at the beginning) to express his positive disposition to a patch generated by his colleague. Thus, $+1$ in a comment indicates positive emotion, while -1 indicates the opposite. We identify presence of $+1$ and -1 in a text and create two binary features: i) hasPlusOne and ii) hasMinusOne.

7.3.3 Algorithm Selection

There are many supervised ML algorithms available [214] for classification problems. Among those, we select Scikit-learn's [214] implementations of following nine ML algorithms as those are popular and frequently used in sentiment/emotion classification.

(a) Adaptive Boosting (AB) [105], (b) Decision Tree (DT) [105, 123], (c) Gradient Boosting Tree (GBT) [124], (d) K-nearest Neighbors (KNN) [105], (e) Naive Bayes (NB) [105, 125], (f) Random Forest (RF) [126], (g) Multilayer Perceptron (MLP) [127], (h) Support Vector Machine with Stochastic Gradient Descent (SGD) [123], and (i) Linear Support Vector Machine (SVM) [105, 128].

7.4 Evaluation

We use *precision* ($\wp$), *recall* ($\Re$), and *F-score* ($\Finv$) to measure the accuracy of emotion detection of `MarValous` for each of the five emotional states (as described in Section 7.2). Given a set $\mathcal{T}$ of texts, *precision* ($\wp$), *recall* ($\Re$), and *F-score* ($\Finv$) for a particular emotional state e is calculated as follows:

$$\wp = \frac{|\,\mathcal{T}_e \cap \mathcal{T}_e'\,|}{|\,\mathcal{T}_e'\,|}, \quad \Re = \frac{|\,\mathcal{T}_e \cap \mathcal{T}_e'\,|}{|\,\mathcal{T}_e\,|}, \quad \Finv = \frac{2 \times \wp \times \Re}{\wp + \Re},$$

where $e \in \{excitation, stress, depression, relaxation, neutral\}$, $\mathcal{T}_e$ represents the set of texts expressing the emotional state e, and $\mathcal{T}_e'$ denotes the set of texts for which `MarValous` identifies the emotional state e.

7.4.1 Dataset

There is only one publicly available dataset where software engineering texts are manually annotated by Islam and Zibran [129] with four emotions *excitation, stress, depression,* and *relaxation*. Emotion-wise number of comments in that dataset are mentioned in the second column of Table 7.4.

Table 7.4: Number of comments in categories of emotions

Emotion	**# of comments annotated by** Islam and Zibran [129]	Novielli et al. [130]	**Total # of Comments**
Excitation	411	1,709	2,120
Stress	252	988	1,240
Depression	289	230	519
Relaxation	227	0	227
Neutral	616	400	1,016
Overall total number of comments:			5,122

The number of comments are not adequate to train and test a ML classifier [215]. Hence, we increase the number of comments by leveraging the dataset created by Novielli et al. [130]. They release a dataset of 4,800 questions, answers, and comments collected from Stack Overflow, which are manually annotated using six basic emotions, namely *love*, *joy*, *anger*, *sadness*, *fear*, and *surprise*. According to emotion classification model in the two-dimensional approach [212] (as seen in Figure 7.1) the emotions *love* and *joy* fall in the first quadrant, emotions *anger* and *fear* fall in the second quadrant and emotion *sadness* falls in the third quadrant. Hence, we select those comments that are annotated with the emotions *love, joy, anger, fear* and *sadness* and assigned those comments the representative emotions of their respective quadrants in which they belong.

In some cases, an expression of surprise can be positive, while in other cases it can convey a negative sentiment/valence [110]. As the polarities along the valence dimension are not defined for the surprised comments in the dataset of Novielli et al., we exclude those surprised comments to minimize ambiguity. Finally, we randomly pick 400 neutral comments of the dataset. For each of the emotions, we mention the number of comments collected from the dataset of Novielli et al. in the third column of Table 7.4. The combined dataset consists of 5,122 comments.

7.4.2 Evaluation of ML Algorithms

Here we seek to identify which ML algorithm shows the best performance on the dataset. We use 10-fold cross-validations to validate each of the algorithms, where the dataset is randomly divided into 10 groups and each of the ten groups is used as test dataset once, while the remaining nine groups are used to train the classifier.

Table 7.5: Comparison of ML algorithms in classification of emotional states

Emotion	Metrics	AB	DT	GBT	KNN	NB	RF	MLP	SGD	SVM
Excit.	$\wp$	80.91%	86.82%	**90.12%**	82.31%	50.92%	80.67%	89.62%	85.88%	89.65%
	$\mathfrak{R}$	88.47%	86.98%	93.13%	89.36%	**98.72%**	90.64%	92.34%	93.60%	93.77%
	$\Finv$	84.48%	86.88%	91.59%	85.68%	67.13%	85.31%	90.96%	89.28%	**91.65%**
Stress	$\wp$	55.08%	63.70%	**79.67%**	75.63%	85.94%	70.42%	74.61%	78.50%	75.81%
	$\mathfrak{R}$	44.79%	54.63%	71.26%	40.33%	22.31%	49.19%	75.55%	69.80%	**76.99%**
	$\Finv$	48.81%	58.64%	75.10%	52.39%	35.26%	57.83%	74.96%	73.44%	**76.32%**
Sad	$\wp$	61.97%	56.01%	**83.97%**	59.26%	20.00%	63.74%	70.83%	74.56%	78.27%
	$\mathfrak{R}$	25.26%	51.25%	**63.66%**	56.52%	00.37%	42.91%	62.25%	62.32%	60.73%
	$\Finv$	34.88%	53.17%	**72.20%**	57.36%	00.72%	51.13%	66.02%	67.27%	68.14%
Relax.	$\wp$	54.20%	64.38%	84.77%	67.41%	00.00%	80.07%	82.22%	80.35%	**86.77%**
	$\mathfrak{R}$	52.07%	59.17%	71.37%	62.72%	00.00%	51.65%	75.62%	76.38%	**77.15%**
	$\Finv$	52.46%	61.22%	77.04%	64.48%	00.00%	62.21%	78.51%	77.29%	**81.29%**
Neutral	$\wp$	58.37%	68.86%	76.83%	59.56%	75.51%	67.54%	86.18%	86.33%	**86.35%**
	$\mathfrak{R}$	77.45%	84.06%	88.94%	84.96%	50.21%	**91.61%**	82.09%	84.38%	87.99%
	$\Finv$	65.42%	75.65%	84.45%	69.91%	59.96%	77.68%	84.00%	85.33%	**87.09%**
Overall average accuracy	$\wp$	62.10%	67.95%	83.07%	68.83%	46.47%	72.49%	80.69%	81.12%	**83.37%**
	$\mathfrak{R}$	57.61%	67.22%	77.67%	66.78%	34.32%	65.20%	77.57%	77.29%	**79.33%**
	$\Finv$	57.21%	67.11%	80.08%	65.96%	32.61%	66.83%	78.89%	78.52%	**80.90%**

Table 7.6: Comparison of features in `MarValous`

Emotion	Metrics	All feat.	η	$\eta+\beta$	$\eta+\gamma$	$\eta+\delta$
Excitation	$\wp$	**89.65%**	88.03%	87.81%	88.42%	88.17%
	$\mathfrak{R}$	93.77%	**94.81%**	94.77%	94.51%	94.34%
	$\Finv$	**91.65%**	91.27%	91.14%	91.35%	91.14%
Stress	$\wp$	**75.81%**	72.36%	72.30%	72.70%	72.47%
	$\mathfrak{R}$	**76.99%**	74.85%	75.13%	76.17%	74.89%
	$\Finv$	**76.32%**	73.41%	73.62%	74.31%	73.52%
Sad	$\wp$	**78.27%**	69.08%	69.71%	70.53%	70.21%
	$\mathfrak{R}$	**60.73%**	56.23%	55.20%	55.72%	56.11%
	$\Finv$	**68.14%**	61.85%	61.32%	62.14%	61.94%
Relaxation	$\wp$	86.77%	88.10%	88.18%	88.05%	**88.96%**
	$\mathfrak{R}$	**77.15%**	73.79%	76.26%	76.66%	75.65%
	$\Finv$	81.29%	79.91%	81.46%	81.56%	**81.57%**
Neutral	$\wp$	86.35%	89.58%	**89.29%**	88.97%	88.53%
	$\mathfrak{R}$	**87.99%**	82.37%	82.17%	83.15%	83.88%
	$\Finv$	**87.09%**	85.76%	85.52%	85.88%	86.11%
Overall average accuracy	$\wp$	**83.37%**	81.43%	81.46%	81.73%	81.67%
	$\mathfrak{R}$	**79.33%**	76.41%	76.70%	77.24%	76.97%
	$\Finv$	**80.90%**	78.44%	78.61%	79.05%	78.86%

Here, feature η ={*unigram* and *bigram*}, β ={*emoticons* and *exclamatory mark*}, γ={*all capital letters*, *elongated word* and *interjection*}

For each of the ML algorithms, we run `MarValous` on the dataset and compute averages of precisions, recalls, and f-measures for *10-fold cross-validations* for each of the emotional states. The computed metrics' values are presented in Table 7.5. For each ML algorithm, the overall average precision, recall, and F-score across all the emotional states are presented in the last three rows. The highest obtained value of a metric in each row is boldfaced for better interpretation of the results.

In Table 7.5, we see SVM obtains the highest F-score values for each of the emotional states except *depression*. For emotions *depression* and *relax*, GBT and SVM obtain the best performances respectively, for all the metrics. For emotions *excitation* and *stress*, GBT shows the best results for metric precision, while NB and SVM obtain the highest recall values for the particular emotions respectively. Although NB performs relatively better in detecting emotions *excitation*, *stress* and *neutral*, which have higher number of comments, it performs very low in detecting *relaxation* and *depression* that have lower number comments. On the other hand, for *neutral* comments, RF obtains the highest recall value, whereas, SVM obtains the highest precision value followed by SGD.

Notably, SGD and MLP algorithms show promising results, although their obtained overall average accuracies' values are lower than the metrics' values obtained by SVM and GBT algorithms. SVM performs the best in terms of overall average precision, recall and F-score (see last three rows of last column in Table 7.5) that indicates the algorithm shows steady performances across all the emotional states. Hence, among the nine ML algorithms, we select this ML algorithm as default for our `MarValous` tool and conduct our subsequent analyses.

7.4.3 Evaluation of Features in `MarValous`

The selection and quality of the features representing each class affect the accuracy and efficiency of classification of ML algorithms [224]. Identifying and removing irrelevant and unnecessary features increase learning accuracy and improve comprehensibility of results. Toward that we measure importance of the seven features of `MarValous` in classifying four emotional states of the comments in our dataset.

First, based on the similarities of the features we divide those features into four disjoint sets: i) η ={*unigram* and *bigram*}, ii) β ={*emoticons* and *exclamatory mark*}, iii) γ={*all capital letters*, *elongated word* and *interjection*} and iv) δ={*plus one* and *minus one*}. Then, for various combinations of those features' sets, we run `MarValous` on our dataset. The combinations of those features and computed results for each of the combinations are presented in Table 7.6.

As seen in Table 7.6, in all the cases, precision, recall and F-score values are always higher when `MarValous` is operated with all features except in four cases. Those four exception cases include: i) the highest recall value obtained using only n-gram feature for comments belong to *excitation* emotion, ii) the best precision and F-score values obtained using the combination of η and δ features for emotion *relaxation*, and iii) the highest precision value obtained using the combination of η and

β features for *neutral* comments. However, those exception cases cannot prevent the combination of all features to obtain the highest overall average accuracies (see last three rows of third column in Table 7.6).

By observing the differences of metrics' value presented in the third and fourth columns of Table 7.6, we see n-gram (η) features i.e., unigram and bigram contribute the most significant part of the accuracies of our tool `MarValous`. Although other features' contributions are not significant, those features play their roles in increasing overall accuracies of our tool `MarValous`. As SVM algorithm with all the seven features has showed the best performance, we have released `MarValous` by setting SVM as its default algorithm while enabling all those features in it.

7.4.4 Comparison with `DEVA`

We compare our tool's accuracies with those of `DEVA` [129]. Until the public release of our `MarValous`, `DEVA` is the only available SE domain specific tool for the detection of the four emotional states that our tool also detects. While `DEVA` uses a lexicon based approach, our `MarValous` relies on ML techniques.

To ensure a fair comparison between the tools, we randomly divide the dataset into two subsets: i) training set contains 70% (i.e., 3,586) comments of the dataset and ii) test set contains remaining 30% (i.e., 1,536) comments. The training set is used to train our ML based tool `MarValous` and the test set is used to compute and compare performances of `DEVA` and `MarValous`.

We separately operate `DEVA` and `MarValous` to detect the emotional states in each of the comments in the test set. Then, we compute the precision ($\wp$), recall ($\Re$), and F-score ($\Finv$) for their detections of each emotional states (i.e., excitation, stress, depression, relaxation, and neutral) as presented in Table 7.7. The overall average precision, recall, and F-score across all the emotional states are presented in the bottom three rows.

We see in Table 7.7 that `MarValous` outperforms the baseline tool `DEVA` in all cases by a large margin except for the recall values of *stressed* and *neutral* comments and precision value of comments belong to *excitation* emotion. `MarValous` maintains a proper balance between precision and recall for each emotional state resulting in higher F-score in each emotional state. Overall, on average, across all the emotions, `MarValous` clearly outperforms the baseline, as it has achieved 19.04% higher precision and 08.19% higher recall values compared to DEVA.

Statistical significance. We apply a non-parametric *McNemar's* test [153, 110] at significance level $\alpha = 0.05$ to verify the statistical significance in the difference of the results obtained by the two tools, `DEVA` and `MarValous`. As the non-parametric test does not require normal distribution of data, this test suits well for our purpose.

We perform a *McNemar's* test on a 2×2 contingency matrix (Table 7.8) derived from the results obtained from the two tools. A number/frequency in a cell is denoted as n_{xy}, where x and y denote

Table 7.7: Comparison between `DEVA` and `MarValous`

Emotions	**Metrics**	**DEVA**	**MarValous**
Excitation	$\wp$	**91.26%**	86.86%
	$\Re$	78.07%	**93.36%**
	$\Finv$	84.15%	**89.99%**
Stress	$\wp$	43.25%	**79.82%**
	$\Re$	**70.32%**	56.13%
	$\Finv$	53.56%	**65.91%**
Depression	$\wp$	36.92%	**90.00%**
	$\Re$	55.81%	**73.26%**
	$\Finv$	44.44%	**80.77%**
Relaxation	$\wp$	75.10%	**75.31%**
	$\Re$	48.35%	**76.08%**
	$\Finv$	58.82%	**75.70%**
Neutral	$\wp$	74.41%	**84.18%**
	$\Re$	**93.38%**	88.08%
	$\Finv$	82.82%	**86.08%**
Overall average accuracy	$\wp$	64.19%	**83.23%**
	$\Re$	69.19%	**77.38%**
	$\Finv$	64.76%	**79.69%**

Table 7.8: Contingency matrix of McNemar's test

# of comments misclassified by both $\mathcal{T}_a$ and $\mathcal{T}_b$	$n_{00} = 146$	$n_{01} = 113$	# of comments misclassified by $\mathcal{T}_b$ but not by $\mathcal{T}_a$
# of comments misclassified by $\mathcal{T}_a$ but not by $\mathcal{T}_b$	$n_{10} = \mathbf{293}$	$n_{11} = 984$	# of comments correctly classified by both $\mathcal{T}_a$ and $\mathcal{T}_b$

Here, $\mathcal{T}_a$ = `DEVA` and $\mathcal{T}_b$ = `MarValous` row and column numbers respectively of a cell in the contingency table. According to the table, MarValous ($\mathcal{T}_b$) performs better than DEVA ($\mathcal{T}_a$) as $n_{10} > n_{01}$. The difference of performances is found to be statistically significant with p-value $= 2.69 \times 10^{-11}$ where $p < \alpha$. We, therefore, conclude that the observed superior performance of our MarValous over DEVA is statistically significant.

7.5 Limitations and Threats to Validity

To have a large dataset, we combine two datasets of Islam and Zibran [129] and Novielli et al. [130], which are created by following *two-dimensional* and *discrete emotional models* respectively. While those two emotional models different from each other, we map categories of emotions from discrete emotional model to the two-dimensional model (see Section 7.4.1), which can be questioned. To validate our mapping process, we randomly pick 20 comments from each category of emotions and manually verify the correctness of mappings.

Although we combine two datasets to increase the size of the used dataset, the number of comments is still low (total 5,122) for training a ML-based classifier. Thus, the generalizability of MarValous can be questioned. However, higher performances of MarValous on combined dataset, which consists of comments collected from two different data sources (JIRA issue comments and Stack Overflow comments), give us confidence that it will perform good enough on different types of software engineering textual artifacts too. Moreover, the newly combined dataset is not perfectly balanced as the emotion relaxation consists of only 04.43% of all comments. However, such imbalance in the dataset could not adversely affect the performance of MarValous.

While there are many ML algorithms available, we have chosen the nine selected algorithms based on their popularity in the community. Moreover, in the selection of those algorithms, we ensure that the selection encompasses varieties of ML algorithms, e.g., Generalized Linear Models (GLM), Support Vector Machine (SVM), Stochastic Gradient Descent (SGD), Nearest Neighbors (NN), ensemble methods and others. There are still scopes to examine performances of other ML algorithms.

Overfitting can be a common threat to any ML based tool. To make sure there is no such overfitting of MarValous accuracies, we use ten-fold cross validations to measure accuracies of the tool. Moreover, we verify the superior performance of MarValous over DEVA using the 30% of the comments, which were never used at the training phase. Such precautions should have limited the threats of overfitting.

The lists of emoticons, interjections and slangs might not have included all available such items. Missing of items from those lists might have restricted the performance of MarValous. Keyword based detection of APIs' names in texts might not be 100% accurate either. However, these limitations also apply to lexicon-based approaches (such as DEVA). We plan to minimize these limitations in our future work.

7.6 Related work

Earlier research related to sentiment analysis in software engineering text mostly used three tools, `SentiStrength` [215], `Stanford NLP` [146], and `NLTK` [46]. `SentiStrength` is the most frequently [110] used tool among those three tools. All of the aforementioned three tools were developed and trained to operate on general purpose texts (e.g., movie and hotel reviews) and they did not perform well enough when operated in a particular domain such as software engineering [109, 128]. Sentiment analyses using these tools are misleading mainly due to the variations in meanings of domain-specific technical terms [110] and high amount of noises (e.g., code snippets, URL and API names) [110, 111, 128].

Blaz and Becker [163] proposed three lexical based methods that consists of a dictionary method, a template method, and a hybrid method for sentiment analysis in job submission tickets related to Information Technology (IT). Those three techniques were tested on formally written texts generated in closed official environment and may not perform well in dealing with *informal* and *noisy* texts frequently used in software engineering artifacts such as commit and code review comments [110]. `SentiStrength-SE` [110], `Senti4SD` [128] and `SentiCR` [111] are three recent sentiment analysis tools especially designed by targeting software engineering text. However, all the aforementioned tools can detect *valence (aka sentiment) only* and cannot other emotional states in more fine-grained levels.

To detect emotions at deeper levels, Murgia et al. [185] constructed a Machine Learning (ML) classifier to identify six emotions *joy, love, surprise, anger, sad,* and *fear* in issue comments related to software development. In another work, Calefato et al. [56] also developed a ML based toolkit named *EmoTxt* to detect those six emotions from technical posts in Stack Overflow. However, neither of these techniques are capable of detecting the emotional states in the well-established bi-directional emotional model that includes both *valence* and *arousal* dimensions [129].

To address the above mentioned issue, Islam and Zibran [129] developed a dictionary and rule-based tool `DEVA` to identify four emotional states *excitation, stress, depression/sadness*, and *relaxation* in the bi-directional emotional model. Our work is inspired by `DEVA`. The lexical approach of `DEVA` is based on limited-size dictionaries and a set of predefined rules. Thus, the performance of `DEVA` is inherently limited by the quality and size of the dictionaries [187]. Moreover, `DEVA` will fail to correctly classify emotions when encountered certain textual structures and content, which are not covered by the rules and dictionaries. To overcome such limitations, in `MarValous`, we have used ML techniques and we have also demonstrated that our ML approach performs substantially better than the lexical approach of `DEVA`.

`TensiStrength` [122] is a tool released before `DEVA`, and it that can detect *stress* and *relaxation* in texts, but cannot capture *excitation* or *depression*. Unlike `MarValous` and `DEVA`, `TensiStrength` is not particularly designed for any specific domain, and thus produce inferior performance compared to DEVA in dealing with software engineering texts [129]. Our MarValous substantially outperforms DEVA as found from the quantitative comparison in this work.

7.7 Summary

In this chapter, we have presented MarValous, which is the first Machine Learning (ML) based tool especially designed for software engineering text to detect individual emotional states *excitation, stress, depression, relaxation* and *neutrality*. By using nine preprocessing steps and seven features, we have developed the tool MarValous that consists of nine popular and effective supervised ML algorithms for emotion/sentiment analysis.

For evaluating the ML algorithms in MarValous, we have combined two existing manually annotated datasets that consists of 5,122 comments collected form JIRA and Stack overflow. From a quantitative evaluation using this dataset, the Linear Support Vector Machine (SVM) algorithm is found to exhibit the best performance (overall average precision 83.37% and recall 79.33%) followed by GBT. The algorithms SGD and MLPC have also showed promising results.

Next, to find an optimal features' set, we have evaluated various combinations of the seven features of MarValous using the highest performed SVM algorithm. We have found that the set of all the seven features have achieved the best overall average accuracies compared to the selected subsets of those features. As SVM algorithm with all the seven features has showed the best performance, we have released MarValous by setting SVM as its default algorithm while enabling all those features in it.

We have also compared the performance of MarValous (with default settings) against the state-of-the-art tool DEVA. From the quantitative comparisons, MarValous is found to achieve 19.04% higher precision and 08.19% higher recall compared to DEVA. A statistical test has confirmed the significant superiority of MarValous over DEVA. The current release of MarValous and combined benchmark dataset are freely available [196] for public use.

Creating a larger dataset of comments annotated with four emotional states remains within our immediate future work. We will put efforts in minimizing the limitations of MarValous to improve its performance. We will also conduct empirical studies of emotions and their probable impacts in software engineering using our MarValous.

Chapter 8

Understanding and Exploiting Developers' Sentimental Variations

In the last three chapters (Chapter 4, Chapter 6, and Chapter 7), we described the software engineering domain specific tools that we have developed for sentiment and emotion analysis to address the sub-problem I. Now, we aim to address sub-problem II (i.e., understanding developers' sentiments in software engineering). In this chapter, we present our fist study to address sub-problem II by conducting a quantitative empirical study of the emotional variations in different types of development activities (e.g., bug-fixing tasks) and development periods (i.e., days and times), in addition to in-depth investigation of emotions' impacts on software artifacts (i.e., commit messages) and exploration of scopes for exploiting emotional variations in software engineering activities.

The chapter is organized as follows. In Section 8.1, we introduce the motivation of the work. The methodology of the work is described in Section 8.2. Results of data analysis are illustrated in Section 9.3. Threats to validity of the work are discussed in Section 8.4. Related work is described in Section 8.5. Finally, we conclude the chapter in Section 8.6.

8.1 Introduction

Emotions are inseparable part of human nature, which influence people's activities and interactions, and thus emotions affect task quality, productivity, creativity, group rapport and job satisfaction [26, 27, 28]. Software development, being highly dependent on human efforts and interactions, is more susceptible to emotions of the practitioners. Hence, a good understanding of the developers' emotions and their influencing factors can be exploited for effective collaborations, task assignments [29], and in devising measures to boost up job satisfaction, which, in turn, can result in increased productivity and projects' success [30].

Several studies have been performed in the past for understanding the role of human aspects on software development and engineering. Some of those earlier studies address *when* and *why* employees get affected by emotions [26, 34, 4, 41, 43], whereas some other work address *how* [60, 61, 62, 63, 31] the emotions impact the employees' performance at work. Despite those earlier attempts, software engineering practices still lack theories and methodologies for addressing human factors such as, emotions, moods and feelings [3, 4]. Hence, the community calls for research on the role of emotions in software engineering [61, 28, 64].

Some software companies try to capture the developers' emotional attachments to their jobs by means of traditional approaches such as interviews and surveys [31]. Capturing emotions with the traditional approaches is more challenging for projects relying on geographically distributed team settings and voluntary contributions (e.g., open-source projects) [33, 34]. Thus, to supplement or complement those traditional sources, software artifacts such as the developers' commit comments/messages have been identified for the extraction of important information including developers' emotional states [34, 4, 41].

In this work, we study the polarity (i.e., positivity, negativity, and neutrality) of emotions expressed in commit messages as posted by developers contributing to open-source projects. In particular, we address the following four research questions.

RQ1: *Do developers express different levels (e.g., high, low) and polarity (i.e., positivity, negativity, and neutrality) of emotions when they commit different types (e.g., bug-fixing, new feature implementation, refactoring, and dealing with energy related concerns) of development tasks?*
— If we can distinguish development tasks at which the developers express high negative emotions, low positive emotions, or an overall low emotional involvements, stipulating measures can be introduced to emotionally influence the emotions of the developers working on those particular types of development tasks resulting in higher success rate.

RQ2: *Can we distinguish a group of developers who express more emotions (positive or negative) in committing a particular type (e.g., bug-fixing) of tasks?*
— Programmers who develop in them positive emotions while carrying out a given development task can be more efficient and quicker in completing the task [63] resulting in reduced software cost. Thus, distinguishing a group of practitioners having positive emotional attachment to a particular task can be useful in effective task assignments.

RQ3: *Do the developers' polarity (i.e., positivity, negativity, and neutrality) of emotions vary in different days of a week and in different times of a day*?
— If we can identify any particular days and times when developers express significant negative emotions, then managers can take motivating steps to boost up the developers positive feelings on those days and times. Guzman et al. [34] reported that commit comments posted on Mondays tend to have more negative emotions. We also want to verify their claim using a substantially larger data-set.

RQ4: *Do the developers' emotions have any impact on the lengths of commit comments they write*?
— Commit messages are pragmatic means of communication among the developers contributing to the same project. Ideally, commit comments contain important information about the underlying development tasks, and the length of developers' work description is an indication of the description quality [225]. If any relationship can be found between the developers' emotional state and

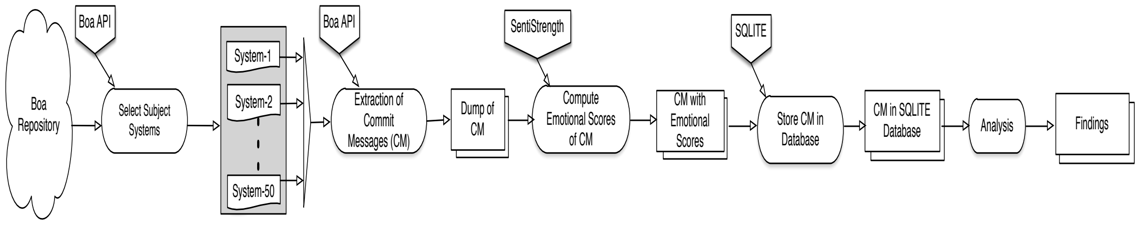


Figure 8.1: Procedural Steps of Our Empirical Study

the lengths of commit comments, then project managers can take steps to stimulate the developers emotional states to get high quality commit comments containing enough contextual information.

8.2 Methodology

To address the aforementioned research questions, we extract emotions from the developers' commit messages using `SentiStrength` [45], which is a state-of-the-art sentiment analysis tool. `SentiStrength` was previously used for similar purposes [42, 4, 43] and was reported to be good candidate for analyzing emotions in commit comments [34]. In the following subsections, we first briefly introduce sentiment analysis with `SentiStrength` (Section 9.2.2) and then, we describe the metrics (Section 8.2.2), tuning of `SentiStrength` (Section 8.2.3) for software engineering context, and data collection approaches (Section 8.2.4) used in our study. The procedural steps of our empirical study are summarized in Figure 14.1.

8.2.1 Sentiment Analysis

Sentiment analysis using `SentiStrength` on a given piece of text (e.g., a commit message) c computes a pair $\langle \rho_c, \eta_c \rangle$ of integers, where $+1 \leq \rho_c \leq +5$ and $-5 \leq \eta_c \leq -1$. Here, ρ_c and η_c respectively represent the positive and negative emotional scores for the given text c.

A given text c is considered to have positive emotions if $\rho_c > +1$. Similarly, a text is held containing negative emotions when $\eta_c < -1$. Note that, a given text can exhibit both positive and negative emotions at the same time, and a text is considered emotionally neutral when the emotional scores for the text appear to be $\langle 1, -1 \rangle$. Further details about the sentiment analysis algorithm of `SentiStrength` and the interpretation of its outputs can be found elsewhere [45].

8.2.2 Metrics

To carry out our analyses for deriving the answers to the research questions, we formulate the following metrics. Given a set C of commit messages, we can obtain two subsets $C+$ and C_- defined as follows:
$C_+ = \{c \quad | c \in C, \rho_c > +1\}$ and $C_- = \{c \quad | c \in C, \eta_c < -1\}$.

Mean Positive Emotional Score for a set C of commit messages, denoted as $\mathcal{P}(C)$, is defined as:

$$\mathcal{P}(C) = \frac{\sum_{c \in C_+} \rho_c}{|C_+|} \tag{8.1}$$

Mean Negative Emotional Score for a set C of commit comments, denoted as $\mathcal{N}(C)$, is defined as follows:

$$\mathcal{N}(C) = \frac{\sum_{c \in C_-} |\eta_c|}{|C_-|} \tag{8.2}$$

Cumulative Emotional Score for a particular commit message c, denoted as $\mathcal{T}(c)$, is defined as follows,

$$\mathcal{T}(c) = \rho'_c + \eta'_c \tag{8.3}$$

where,

$$\rho'_c = \begin{cases} \rho_c, & \text{if } \rho_c > +1. \\ 0, & \text{otherwise.} \end{cases} \qquad \eta'_c = \begin{cases} |\eta_c|, & \text{if } \eta_c < -1. \\ 0, & \text{otherwise.} \end{cases}$$

8.2.3 Tuning of SentiStrength

The sentiment analysis tool `SentiStrength` was reported to have 60.7% precision for positive texts and 64.3% for negative texts [45]. To the best of our knowledge, all such sentiment analysis tools including `SentiStrength` are highly dependent on the polarities of individual words in a given text in computation of its emotional scores. `SentiStrength` was originally trained on documents on the social web. In a technical field such as software engineering, commit messages include many keywords which have polarities in terms of dictionary meanings, but do not really express any emotions in their technical context. For example, 'Super', 'Support', 'Value' and 'Resolve' are English words with known positive emotions, while 'Dead', 'Block', 'Default', and 'Garbage' are known to have negative emotions, but neither of these words really bear any emotions in software development artifacts. Those are simply some domain specific technical terms with especial contextual meanings.

To save `SentiStrength`'s computation of emotional scores from being mislead by such technical terms, we tune the tool for application in our software engineering context. Based on our manual investigation, experience, and literature review [41, 43], we identify a total of 49 terms, which can be misinterpreted by `SentiStrength`. These misleading terms are: *'Arbitrary', 'Block', 'Bug', 'Conflict', 'Constraint', 'Corrupt', 'Critical', 'Dynamic', 'Dead', 'Death', 'Default', 'Defect', 'Defensive', 'Disabled', 'Eliminate', 'Error', 'Exceptions', 'Execute', 'Failure', 'Fatal', 'Fault', 'Force', 'Garbage', 'Greater', 'Inconsistency', 'Interrupt', 'Kill', 'Like', 'Obsolete', 'Pretty', 'Redundant', 'Refresh', 'Regress', 'Resolve', 'Restrict', 'Revert', 'Safe', 'Security', 'Static', 'Super', 'Support', 'Success', 'Temporary', 'Undo', 'Value', 'Violation', 'Void', 'Vulnerable' and 'Wrong'.*

SentiStrength provides the flexibility to modify its existing lexicons' emotional interpretation to customize it for a target context (i.e., software engineering, in this work). For our purpose, we neutralize SentiStrength's interpretation of the aforementioned technical jargons, as such was also suggested in earlier studies in the area [41, 43].

Table 8.1: Examples of commit comments and computation of their emotional scores

Boa Proj. ID	**Commit/Revision ID: Commit Comment (c)**	**Emo. Score**		
		ρ_c	η_c	$\mathcal{T}(c)$
12562083	7519717434bb0ae5fad5329885bd184e7b502d27: Fixes #1721 Committing work by Arnfried (EXCELLENT!)	5	-1	5
689344	2605951f8b73a963beb01b3806b3ad43ce638848: Don't save mute setting. Extremely annoying to start with lack of audio and have no idea what causes it	1	-5	5
11814891	01845191185d0a14960f1542ac77f512f8749514: a bit more detailed test; hope this avoids some reflection searches in FF emulation and makes the monster faster in special situations	3	-2	5
689344	00058618c9c3f1f9fc4d9310012a0d1881c0c940: (RMenu) RMenu refactor - have function pointers for menu struct	1	-1	0

Having SentiStrength tuned according to the procedure described above, we manually verify the impact of the tuning using a random sample of 200 commit messages extracted from Boa [131], and we found a 26% increase of precision (checked by comparing SentiStrength's computation of emotional polarities with subjective human interpretation over each of the 200 commit messages). Thus, for our work, we use this improved instance of SentiStrength tuned for use in software engineering context.

8.2.4 Data Collection

We study commit messages for open-source projects obtained through Boa [131]. Boa is a recently introduced infrastructure with a domain specific language and public APIs to facilitate mining software repositories. We use the largest (as of February 2016) data-set from Boa, which is categorized as "full (100%)" and consists of more than 7.8 million projects collected from GitHub before September 2015.

From this large data-set, we select the top 50 projects having the highest number of commits. We study all the commit messages in these projects, which constitute 490,659 commit comments. Associated information such as, committers, commit timestamps, types of underlying work, revisions and project IDs are kept in a local relational database for convenient access and query. For each of the commit messages, we compute the emotional scores using the tuned SentiStrength tool. Table 8.1 shows some examples of emotional and neutral commit comments in our dataset and computation of their emotional scores.

8.3 Analysis and Findings

The research questions *RQ1, RQ2, RQ3* and *RQ4* are respectively addressed in Section 8.3.1, Section 8.3.2, Section 8.3.3, and in Section 8.3.4.

8.3.1 Emotional Variations in Different Task Types

We investigate whether developers' emotions vary based on their involvements in four different types of software development tasks: (a) bug-fixing tasks, (b) new feature implementation, (c) refactoring, and (d) energy-aware development. We consider that the first three types of tasks mentioned above are self-explanatory. The fourth one (i.e., energy-aware development) deals with software issues with consumption of energy, measured in terms of usage of resources such as processing power and memory. Energy-aware development is a recent important topic in the area of green computing research. Categorization of development tasks in this manner are also found in earlier studies [226, 36, 227] in software engineering research.

Task-based Characterization of Commits: To distinguish commits dealing with *bug-fixing* tasks, we rely on Boa's public APIs, which readily indicate whether a commit message is associated with bug-fixing task, or not.

To identify *energy-aware* commit messages, we select a list of keywords and search those keywords in commit messages. A commit message will be considered as energy-aware commit, if the commit message contains any of the selected keywords. The identified keywords are: **energy consum*, *energy efficien*, *energy sav*, *save energy*, *power consum*, *power ecien*, *power sa*, *save power*, *energy drain*, *energy leak*, *tail energy*, *power efficien*, *high CPU*, *power aware*, *drain*, *no sleep*, *battery life* and *battery consum**. The character '*' in each keyword works as a wildcard, i.e., a query will select those commits messages, which contain at least one of these keywords, regardless of the beginning or the end of the commit message. Note that, these keywords were also used in earlier studies [36, 228, 229, 227] for similar purposes.

To recognize commit messages dealing with *new feature implementation* and *refactoring* tasks, we select those keywords, which were used by Ayalew and Mguniin [226] in their work. Keywords **add* and *new feature** are used to categorize commit messages, which are related to new feature development. And **refactor* and *code clean** keywords are used to distinguish those commit messages, which are posted by developers during code refactoring tasks. Note that, a developer may perform refactoring while fixing a bug. Thus, a commit message can be characterized relevant to more than one categories of tasks.

Investigation: The numbers of commit messages found relevant to each of the four categories of development tasks are presented in the second column from left in Table 8.2. The boxplot in Figure 9.2 presents the distribution of mean positive, negative, and cumulative emotional scores in each type

Table 8.2: Commits of different task categories and *MWW* tests between positive and negative emotions in them

Task Categories	# of Commits	P-value	Significant?
Bug-Fixing	117,249	0.03288	Yes ($P < \alpha$)
New Feature	89,019	0.00256	Yes ($P < \alpha$)
Refactoring	5,431	0.04006	Yes ($P < \alpha$)
Energy-Aware	182	0.39743	No ($P > \alpha$)

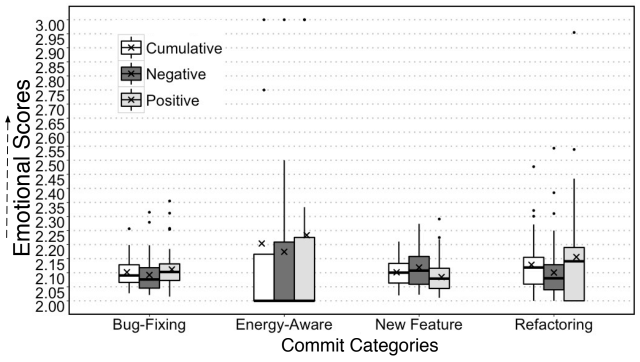


Figure 8.2: Distribution of mean positive, negative, and cumulative emotional scores in commits messages dealing with different types of tasks

of task for each of the 50 projects. An 'x' mark in a box in the boxplot indicates the mean emotional scores over all the projects.

As observed in Figure 9.2, emotional scores (positive, negative and cumulative) for energy-aware commit messages are much higher than those in commit messages for three other tasks, and there is not much variations in the emotional scores among these three tasks. To verify the statistical significance of these observations, we conduct *Mann-Whitney-Wilcoxon (MWW)* tests [132] (with $\alpha = 0.05$) between the distributions of mean cumulative emotional scores in commit messages for each possible pair of development tasks. The results of the *MWW* tests are presented in Table 8.3. The P-values reported by the tests, as compared with α, suggest statistical significance of our observations.

Table 8.3: *MWW* tests between cumulative emotional scores of commit messages dealing with different types of tasks

Task Categories	Bug-Fixing	Refactoring	New Feature	Energy-Aware
Bug-Fixing	-	0.75656	0.89656	0
Refactoring	0.75656	-	0.71884	0
New Feature	0.89656	0.71884	-	0
Energy-Aware	0	0	0	-

Again, looking at Figure 9.2, we see that the commit messages, which are posted during the new features implementation tasks, show more negative emotions than positive ones. Opposite observations are evident for commit messages for three other types of tasks. To verify the statistical significance of our observations in the variations of polarity (positivity and negativity) of emotions, for each of the four types of development tasks, we separately conduct *MWW* tests between the mean positive and negative emotional scores of commit messages. The results of the *MWW* tests are presented in the right-most two columns in Table 8.2. The P-values of tests, as compared with α, suggest statistical significance of our observations for bug-fixing, new feature implementation and refactoring tasks, but not for the *energy-aware* development tasks.

Based on our observations and statistical tests, we derive the answer to the research question *RQ1* as follows:

Ans. to RQ1: *Developers express significantly high positive and negative emotions almost equally in committing energy-aware tasks. For bug-fixing and refactoring tasks, positive emotions are significantly higher than negative emotions. And surprisingly, for new feature implementation tasks, negative emotions are significantly higher than positive polarity.*

8.3.2 Emotional Variations in Bug-Fixing Tasks

It is natural that different developers have different expertise, comfort-zones, and interests with respect to types of tasks. The research question *RQ2* addresses the possibility of distinguishing a set of developers who particularly express positive emotions at the particular type of task at hand. In addressing the research question *RQ2*, we choose the *bug-fixing* tasks as a representative to any particular type of tasks and continue as such.

Across all the projects, we distinguish 20 developers, who are the authors of the bug-fixing commit messages having the highest positive mean emotional scores. Let $\mathcal{D}_p$ denote the set of these 20 developers. Similarly, we form another set $\mathcal{D}_n$ consisting of 20 developers, who are the authors of bug-fixing commit comments having the highest negative mean emotional scores. By the union of these two sets, we obtain a set $\mathcal{D}$ of 30 developers who are authors of bug-fixing commits with the highest mean positive or negative emotional scores. Mathematically, $\mathcal{D} = \mathcal{D}_p \cup \mathcal{D}_n$.

These 30 developers are the authors of 112,462 commits messages among which 32,088 are bug-fixing commits. For each of these 30 developers, we compute a ratio $\mathcal{R}(d)$ as follows:

$$\mathcal{R}(d) = \frac{\mathcal{P}(C_d)}{\mathcal{N}(C_d)},\ where,\ d \in \mathcal{D} \tag{8.4}$$

Here, C_d denotes the set of bug-fixing commit comments posted by developer d. Notice that, for a particular developer d, the ratio $\mathcal{R}(d)$ close to 1.0 indicates that the positive and negative emotions are almost equal for the developer d. If $\mathcal{R}(d)$ is much higher than 1.0, the developer d shows more positive emotions at bug-fixing tasks compared to negative emotions. The opposite holds when $\mathcal{R}(d)$

is much lower than 1.0. However, a threshold scheme seems necessary to determine when the value of $\mathcal{R}(d)$ can be considered significantly close to or distant from 1.0.

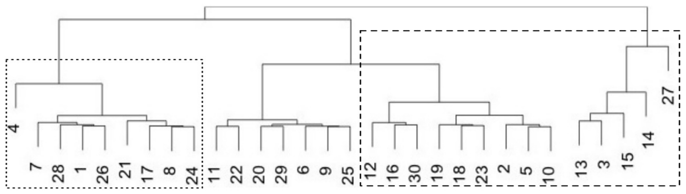


Figure 8.3: Hierarchical agglomerative clustering of 30 developers enumerated as $1, 2, 3, \ldots, 30$

Clustering Analysis: Instead of setting an arbitrary threshold, we apply unsupervised *Hierarchical Agglomerative Clustering* for partitioning the values of $\mathcal{R}(d)$. The dendrogram produced from this clustering is presented in Figure 8.3. In the dendrogram, we identify three major clusters/groups, two marked (by us) with dotted rectangles and the third left unmarked in the middle. This middle cluster, denoted as G_b, represents the set of those developers, who equally express positive and negative emotions during bug-fixing. We have, $0.992 \leq \mathcal{R}(d) \leq 1.0, \forall d \in G_b$.

The set of developers who are included in the right-most cluster exhibit more positive emotions compared to negative emotions during bug-fixing. Let G_p denote the cluster of these developers. Here, $1.005 \leq \mathcal{R}(d) \leq 1.178, \forall d \in G_p$. The set of developers who render more negative emotions towards bug-fixing belong to left-most cluster, denoted as G_n. We have, $0.919 \leq \mathcal{R}(d) \leq 0.982, \forall d \in G_n$.

Table 8.4: *MWW* tests over $\mathcal{R}(d)$ scores of commit messages written by developers in each cluster

Cluster	G_p	G_n	G_b
***P*-values**	0.00798	0.0268	0.26109
Significant?	Yes ($P < \alpha$)	Yes ($P < \alpha$)	No ($P > \alpha$)

Statistical Significance: For each of the three clusters, we separately conduct *MWW* tests between the mean positive and negative emotional scores of the commit messages to verify the statistical significances of their differences. The results of the separate *MWW* tests (with $\alpha = 0.05$) over each of the clusters are presented in Table 8.4. The P-values in Table 8.4 indicate statistical significance in the differences in positive and negative emotions for clusters G_p and G_n. As expected, no such significant difference found for the cluster G_b as in this cluster, positive and negative emotions are expressed equally. Thus, our clustering of the developers appears to be accurate with statistical significance. Hence, we answer the research question *RQ2* as follows:

Ans. to RQ2: *We have been able to distinguish sets of developers who show either high positive or high negative emotions in bug-fixing commit messages while some other developers are found to express both positive and negative emotions almost equally. The same approach can be applied to distinguish such groups of developers for other types of development tasks.*

8.3.3 Emotional Variations in Days and Times

For each of the projects, we group all the commit messages into seven disjoint sets in accordance with the days of the week those are committed.

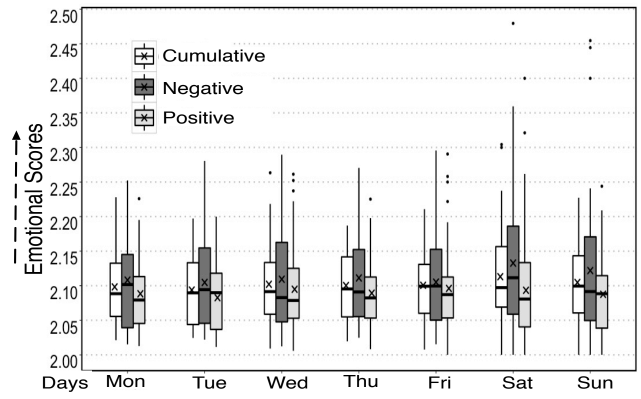


Figure 8.4: Distribution of mean positive, negative, and cumulative emotional scores in commit comments posted in different days of week

Table 8.5: *MWW* tests over cumulative emotional scores of commit messages written in different days of week

Day	Sat	Sun	Mon	Tue	Wed	Thu	Fri
Sat	-	0.44	0.23	0.11	0.33	0.41	0.35
Sun	0.44	-	0.71	0.42	0.77	0.98	0.84
Mon	0.23	0.71	-	0.68	0.79	0.71	0.84
Tue	0.11	0.42	0.68	-	0.55	0.41	0.49
Wed	0.33	0.77	0.79	0.55	-	0.83	0.96
Thu	0.41	0.98	0.71	0.41	0.83	-	0.86
Fri	0.35	0.84	0.84	0.49	0.96	0.86	-

Figure 8.4 plots the average (over each project) positive, negative, and cumulative emotional scores in commit messages posted in different days in a week. Among *all the seven days* of a week, negative emotions appear to be slightly higher in commit messages posted during the weekends (i.e., Saturday and Sunday) than those posted in weekdays (i.e., Monday through Friday). Not much differences are visible in the emotional scores for commit messages posted in the *five weekdays*. *MWW* tests (with $\alpha = 0.05$) between the distributions of emotional scores in each possible pair of the days of a week suggest no statistical significance in the differences of emotions. P-values of the *MWW* tests are presented in Table 8.5. As can be seen in Table 8.5, for all values of P's, $\alpha < 0.11 \leq P$.

To study the relationship between developers emotions and times of a day when commit comments are posted, we divide the 24 hours of a day in three periods (a) 00 to 08 hours as *before working hours*, (b) 09 to 17 hours as regular *working hours* and (c) 18 to 23 hours as *after working hours*. Then for each project, we again organize the commit messages into three disjoint sets based on their timestamps of posting.

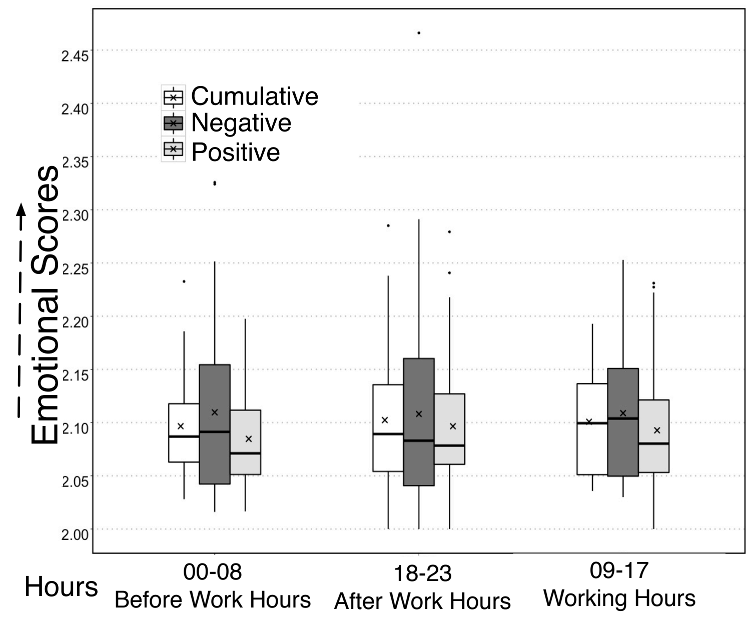


Figure 8.5: Distribution of mean positive, negative, and cumulative emotional scores in commit comments posted in different periods of day

Figure 8.5 presents the mean (over each project) positive and negative emotional scores (computed using Equation 8.1 and Equation 8.2) in commit messages posted in these three periods. Again, in Figure 8.5, we do not see much variations in the emotional scores of commit messages posted at different periods. *MWW* tests (with $\alpha = 0.05$) between the distributions of mean positive and negative emotional scores in each possible pair of the periods indicate no statistical significance in their differences. P-values of the *MWW* tests are presented in Table 8.6. Hence, we derive the answer to the research question *RQ3* as follows:

Ans. to RQ3: *There is no significant variations in the developers' emotions in different times and days of a week.*

Table 8.6: *MWW* tests over cumulative emotional scores of commit messages written in different times of a day

Hours in a Day	00-08	09-17	18-23
00-08	-	0.59612	0.84148
09-17	0.59612	-	0.85716
18-23	0.84148	0.85716	-

8.3.4 Emotional Impacts on Commit Lengths

To investigate the existence of any relationship between emotions and lengths of commit messages, across all the 50 projects, we distinguish 141,033 commit comments, which are one to 50 words in length having cumulative emotional scores (computed using Equation 8.3) higher than one. For each project, we organize these emotional commit messages into four disjoint groups based on their lengths as shown in Figure 8.3.4, which plots the mean (over each project) cumulative emotional scores of commit messages in the four groups. As seen in the figure, the emotional scores are strictly higher for the groups with lengthier commit messages.

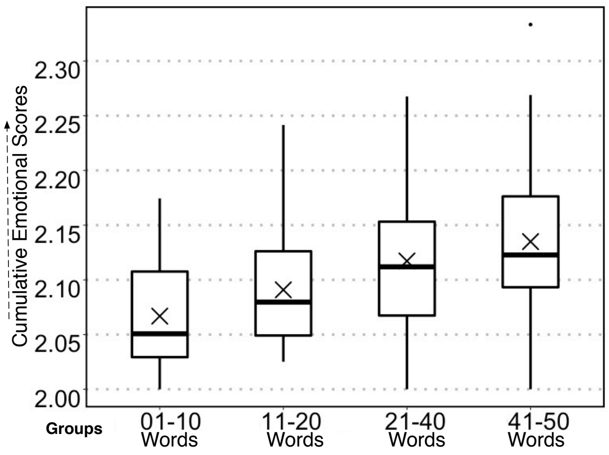


Figure 8.6: Distribution of mean cumulative emotional scores of commit comments in groups of different lengths

Table 8.7: Number of commit messages with different lengths (in words) and cumulative emotional scores

	Commit Length	**# of Commits Comments with $\mathcal{T}(c)$ =**						
		02	**03**	**04**	**05**	**06**	**07**	**08**
Groups	**01-10**	46,486	2,734	2,558	245	28	12	3
	11-20	42,144	3,967	4,627	876	145	16	1
	21-40	22,732	2,633	5,008	1,155	203	31	2
	41-50	3,255	409	1,275	399	84	5	0

Table 8.7 presents the frequencies of commit messages in the four groups and having different cumulative emotional scores. A *Chi-squared* [132] test ($P = 2.2 \times 10^{-16}, \alpha = 0.05$) also strongly indicates statistical significance of the relationship between emotional scores and commit lengths. Next, we verify the significance of the *direction* of relationship (i.e., if one increases or decreases with the increase of another).

Fitting of a *Generalized Linear Model* [132] on the emotional score and length of every emotional commit message confirms (with $\beta = +0.01134, P = 2 \times 10^{-16}, \alpha = 0.001$) the positive correlation between emotional scores and commit lengths. Based on the analyses, we now derive the answer to the research question *RQ4* as follows:

Ans. to RQ4: *Developers' emotions have statistically significant impacts on the lengths of commit messages they write. Developers post longer commit comments when they are emotionally active.*

8.4 Threats to Validity

In this section, we discuss the limitations of our work, the threats to the validity of our findings, and our attempts to minimize those threats.

Internal Validity: The *internal validity* of our work depends on the accuracy of the tool's computation of emotional scores. `SentiStrength` was reported to be effective in sentiment analysis [45] and suitable for extraction of emotions from commit comments [34]. `SentiStrength` has relatively high accuracy compared to other tools of its kind and thus `SentiStrength` was used for sentiment analysis in earlier work in software engineering research [36, 42, 34, 4, 43]. Moreover, for use in our work, we increased it accuracy in emotion extraction by 26% through tuning the tool for application in software engineering context (Section 8.2.3).

Nevertheless, the tuned tool is not 100% accurate in determining emotional polarities of commit messages, and it was not possible to perform manual sanity check by going through each of the 490,659 commit messages included in our work. We are aware of this threat, although we minimized it by contextual customization of `SentiStrength`.

Construct Validity: The choice of the 30 developers in examining the relationships between emotions and bug-fixing tasks (Section 8.3.2) can be questioned. Note that, these 30 developers are the authors of more than *112 thousand* commit messages (22.85%), which is a large sample of data for dependable analysis. The objective was to check if it was possible to distinguish a group of developers who are emotionally more active towards a particular type of task. If we chose a fair number of developers other than our choice of 30, we would still be able to distinguish a set of target developers. In that case, the size of the set of developers might be different from what we found using the 30 developers, but this does not invalidate the findings of the work.

One may also question the validity of our categorization of the developers' commits in different days and periods (Section 8.3.3), considering the possibility that the projects and developers may be physically located at different geographic locations and time-zones. However, as we found, most (86%) of the commits are posted in regular weekdays. Moreover, the majority (58%) of the commit messages are written in regular working hours while 31% and 11% are found to have been posted respectively in before and after regular working hours. The proportions of commits at different days and periods suggest correctness of the categorization.

In the analysis of the emotional impacts on the lengths of commit messages (Section 8.3.4), we excluded commit messages longer than 50 words, because we observed that commit messages of larger lengths include copy-pasted content such as, SQL statements and code snippets. Such contents are not directly created or typed by the committer and thus are unlikely to reflect his or her emotions.

For the statistical tests of significance in the variations of different distributions, we used the *Mann-Whitney-Wilcoxon (MWW)* test [132]. The *MWW* test is a non-parametric test, which do not require the data to have normal distribution. Since the data in our work do not conform to normal distribution, this particular test suits well for our purpose. Moreover, the significance level α set to 0.05, which is a widely adopted value for this parameter that enables 95% confidence in the results of the *MWW* tests.

External Validity: The findings of this work are based on our study on more than 490 thousand commit messages across 50 open-source projects. This large data-set yields high confidence on the generalizability of the results.

Reliability: The methodology of data collection, analysis, and results are well documented in this chapter. The sentiment analysis tool, `SentiStrength` [45] is freely available online and projects studied in this work are also freely accessible through `Boa` [131]. Hence, it should be possible to replicate this study.

8.5 Related Work

To explore the impacts of emotions on the debugging performance of software developers, Khan et al. [61] used high-arousal-invoking and low-arousal-invoking movie clips to trigger different levels of emotions in developers before having them perform some debugging tasks. However, they did not employ any measurement to extract and quantify the developers emotional states, and relied on the assumption that watching those movie clips would induce different levels of emotions in the developers. Lesiuk [62] recruited 56 software engineers to understand impact of emotions on software design performance. In her work, music was played to arouse developers' positive emotions. The participants self-assessed their emotional states and design performance. Similarly, self-assessment of emotional states were also used in the studies of Wrobel [31] and Graziotin et al. [60].

While the human participants themselves can be expected to accurately report their emotional states, such self-assessment based approaches suffer from the possibility that the participants might be uncomfortable in disclosing their negative emotional states. Biometric measurements such as multi-sensor inputs [32], audio and video processing [230] do not suffer from such difficulties but they can be logistically expensive and difficult for regular use at workplace without disrupting the natural workflow of the practitioners. Both the self-assessment-based and biometric approaches for identification of emotions are difficult (if not impossible) to apply for geographically distributed teams and for extraction of emotions from software artifacts of already completed parts of projects.

Note that, unlike our work, all of the research mentioned above, focused on understanding the *overall* emotional impacts over human performance and indicated positive correlation between them. In contrast, ours include a deeper analysis exploring the impacts and scopes for exploitation of emotions extracted from textual software artifacts such as commit messages. Several other studies also identify developers' emotions from textual software artifacts. In such a study, Murgia et al. [63] reported that issue reports, which express positive emotions take less time to be resolved. They used human raters to identify emotions in issue reports, and thus their work is subject to human errors. Unlike theirs, using an automatic tool `SentiStrength`, we identify emotions in a significantly larger number of commit messages. The automatic tool, `SentiStrength` was also used in the studies of Guzman and Bruegge [4], Tourani et al. [43], Garcia et al. [42], Guzman et al. [34], and in the work of Chowdhury and Hindle [36]. But none of these work tuned the tool before application in software engineering context, as we did in our work.

Guzman and Bruegge [4] identified emotions in collaboration artifacts to relate them with different development topics. In a separate study, using `SentiStrength`, Guzman et al. [34] extracted emotions expressed in 60,425 commit messages and reported that commit comments written on Mondays tend to have more negative emotions compared to Sunday, Tuesday, and Wednesday. However, from the investigation of the same phenomenon using a substantially larger dataset of 490,659 commit messages, our study does not identify any statistically significant variations of emotions in commit comments posted in different days of a week.

Using a Natural Language Toolkit (`NLTK`), Pletea et al. [41] mined developers' emotions from 60,658 commits and 54,892 pull requests for `GitHub` projects. They analyzed emotional variations in discussions on different topics and reported to have found higher negative emotions in security-related discussions in comparison with other topics. While their objective, approach as well as source of emotional content and method of emotion extraction were different from our work, ours includes a deeper and larger analysis based on a larger number of commit messages and diverse aspects of emotional implications.

Using `SentiStrength`, Tourani et al. [43] extracted emotions from emails of both developers and system users. They observed the differences of emotional expressions between developers and users of a system. Using the same tool, Garcia et al. [42] extracted developers' emotions from their email contents to analyze any relationships between developers' emotions and their activities in an open source software projects. Although the studies of Tourani et al. [43] and Garcia et al. [42] also used the same sentiment analysis tool we used, the source of their emotional content are different and the objectives of those work are also orthogonal to ours.

8.6 Summary

In this chapter, we have presented a quantitative empirical study on the characteristics and impacts of emotions extracted from developers' commit messages. We have studied more than 490 thousand commit comments over 50 open-source projects. Although the majority (65%) of the commit messages are found to be neutral in emotion, surprisingly, positive emotions are found in relatively much smaller portion (13%) of the commit comments than the commits (22%) containing negative emotions.

In our study, we found that the polarities of the developers' emotions significantly vary depending on the type of tasks they are engaged in. The developers express equally high positive and negative emotions in committing in energy-aware tasks compared to other tasks. With respect to the polarities of commit messages, positive emotions are found to be significantly higher than negative emotions in commits for *bug-fixing* and *refactoring* tasks. Surprisingly, the opposite scenario is found for *new feature implementation* tasks.

We also found significant positive correlation between the lengths of commit messages and the emotions expressed in them. When the developers remain emotionally active, they tend to write longer commit comments. However, we did not find any *significant* variations in the developers' emotions in commit messages posted in different times and days of a week.

Based on emotional contents in commit messages, we have also been able to distinguish a group of developers who express more positive emotions at bug-fixing commit messages, another group with the opposite trait, and a third group of developers who equally render both positive and negative emotions at bug-fixing activities. Same approach can be applied for other types of tasks to distinguish potential developers for improved tasks assignment.

The findings from this work are validated in the light of statistical significance. Although more experiments can be conducted to verify or confirm the findings, the results from this study significantly advance our understanding of the impacts of emotions in software development activities and artifacts, and we exemplify how emotional awareness can be exploited in improving software engineering activities.

For automatic computation of emotional polarities in commit messages, we have used a state-of-the-art tool, `SentiStrength`, while alternatives exist. Moreover, before applying the tool, we tuned it for our work in the context of software engineering. In future, we plan to replicate this study using other tools and subjects to further validate the findings of this study. We also have plan to conduct more studies on the impacts of emotions extracted from diverse artifacts including program comments, development forums and email groups.

Chapter 9

Sentiment Analysis in Commit Messages of Buggy Code

In the last chapter (Chapter 8), we have presented a quantitative empirical study on the characteristics and impacts of emotions of developers. In this chapter, we present a study of emotional variations in bug-introducing and bug-fixing commit messages that helps in understanding of the extent to which emotions affect tasks resulting in software bugs.

Rest of the chapter is organized as follows. In Section 9.1, we introduce the motivation of the work. The methodology of the work is described in Section 9.2. Data analysis and findings are illustrated in Section 9.3. Threats to validity of the work are discussed in Section 9.4. Related work is described in Section 9.5. Finally, we conclude the chapter in Section 9.6.

9.1 Introduction

Software developers, being humans, are affected by their emotions, which influence their activities and interactions. Thus, emotions affect task quality, productivity, creativity, group rapport and job satisfaction [26, 28]. Several studies have been performed in the past for understanding roles of emotions on software development activities. Some of those earlier studies address *when* and *why* employees get affected by emotions [26, 34, 4, 41, 43].

Some other studies determine correlations between various job performance factors (e.g., productivity, quality and efficiency) and emotions that developers experience during development activities [60, 37, 38, 61, 62, 63, 31]. In addition, a few studies have *successfully* used emotion as a factor in prioritizing applications' features to develop [133] and in predicting qualities of developers' interactions (e.g., asking questions and answering) in technical forums [134, 135, 136] and bug severity [231].

Considering above studies, it deems emotions can be an influential factor to be used in complex machine learning and deep learning systems to predict bugs (i.e., buggy commits) in software. However, before using emotions to predict bugs, we need to empirically evaluate of such possibility in the context. Towards this goal, in this work, we study the polarity (i.e., positivity, negativity, and neutrality) of emotions expressed in two types of commit messages, (i) *bug-introducing* and (ii)

Table 9.1: Subject Systems

Systems	**Lang.**	**Domain**	**BIC***	**BFC$^+$**
Netty	Java	Network	8,745	8,739
Presto	Java	SQL	2,963	2,963
Facebook-android-SDK	Java	Social Network	740	740

*BIC=Bug-introducing commits, $^+$BFC=Bug-fixing commits

bug-fixing, which are posted by developers contributing to open-source projects. In particular, we address the following two research questions.

RQ1: *Do developers express different levels (e.g., high, low) and polarity (i.e., positivity, negativity, and neutrality) of emotions in bug-introducing and bug-fixing commits?*

— If we can identify developers express higher level emotions (either positive or negative) during commits, which cause bugs in software compared to other commits (e.g., bug-fixing commits) then expressed emotional levels can be used as a feature to predict bugs in commits. In addition, here we also want to verify the finding of Islam and Zibran [38] where they claim positive emotions are significantly higher in bug-fixing commits compared to negative emotions.

RQ2: *Do the developers' polarity (i.e., positivity, negativity, and neutrality) of emotions vary in different times of a day in bug-introducing and bug-fixing commits*?

— Here we conduct a deeper analysis by grouping bug-introducing and bug-fixing commits according to their commit timestamps. If we can identify any particular times in a day when developers express significant negative emotions, then managers can take required steps to uplift the developers positive feelings at those times.

9.2 Methodology

The procedural steps of our empirical study are summarized in Figure 14.1. To address the aforementioned research questions, we extract bug-introducing and bug-fixing commit messages from three selected projects. Then, we compute emotional scores of commit messages using the tool `SentiStrength-SE` [110, 109], which is the first domain specific sentiment analysis tool for software engineering texts. Finally, to answer the research questions we conduct statistical analyses. In the following we briefly describe the procedures of data collection, computation of emotional scores and how we conduct data analysis.

9.2.1 Data Collection

We collect three open-source projects from GitHub listed in Table 14.1 along with other information related to those projects. Those three projects are also used in other studies [232, 233]. The bug-

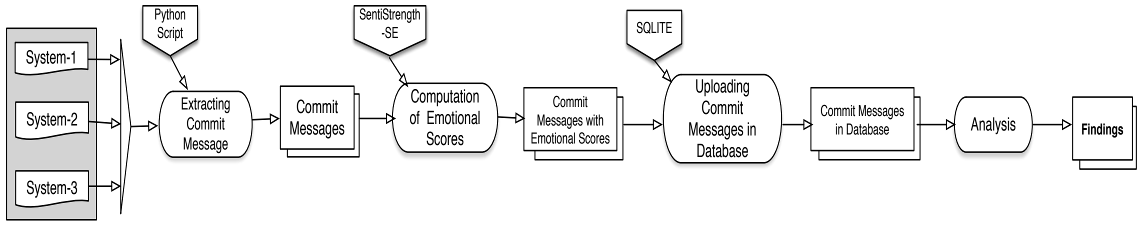

Python Script
SentiStrength -SE
SQLITE
System-1
System-2
System-3
Extracting Commit Message
Commit Messages
Computation of Emotional Scores
Commit Messages with Emotional Scores
Uploading Commit Messages in Database
Commit Messages in Database
Analysis
Findings

Figure 9.1: Procedural steps of our empirical study

introducing and bug-fixing commit messages of those projects are identified in an earlier study [232] that we reuse in this study.

We study all those two types of commit messages in these projects, which constitute 24,890 commit comments. Associated information such as, committers, commit timestamps, revisions and project names are kept in a local relational database for convenient access and query.

9.2.2 Sentiment Analysis

For each of the commit messages, we compute the emotional scores. To do that we use the tool `SentiStrength-SE` [110]. Sentiment analysis using `SentiStrength-SE` on a given piece of text (e.g., a commit message) c computes a pair $\langle \rho_c, \eta_c \rangle$ of integers, where $+1 \leq \rho_c \leq +5$ and $-5 \leq \eta_c \leq -1$. Here, ρ_c and η_c respectively represent the positive and negative emotional scores for the given text c.

A given text c is considered to have positive emotions if $\rho_c > +1$. Similarly, a text is held containing negative emotions when $\eta_c < -1$. Note that, a given text can exhibit both positive and negative emotions at the same time, and a text is considered emotionally neutral when the emotional scores for the text appear to be $\langle 1, -1 \rangle$. Further details about the sentiment analysis algorithm of `SentiStrength-SE` and the interpretation of its outputs can be found elsewhere [110].

9.2.3 Statistical Measurements

To verify the statistical significance of emotional variances in bug-introducing and bug-fixing commit messages, we apply the statistical *Mann-Whitney-Wilcoxon (MWW)* test [234] at the significance level $\alpha = 0.05$. The non-parametric *MWW* test does not require normal distribution of data, and thus it suits well for our purpose. To measure the effect size, we compute the non-parametric effect size *Cliff's delta d* [234]. We consider *significant difference* exists between distributions of emotional scores in bug-introducing and bug-fixing commits if p-value of a *MWW* test is found to be less than α and *Cliff's delta d* value is not negligible (i.e., $|d| > 0.15$).

9.3 Analysis and Findings

The research questions *RQ1* and *RQ2* are respectively addressed in Section 9.3.1 and Section 9.3.2.

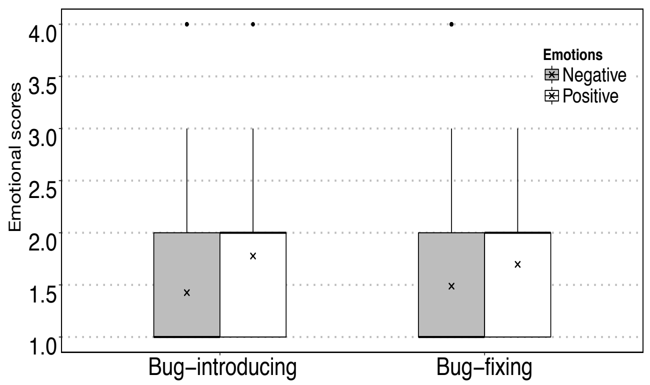


Figure 9.2: Distributions of positive and negative emotional scores in commits messages dealing with bug-introducing and bug-fixing tasks

9.3.1 Overall Emotional Variations

Emotional variations in commit types: The boxplot in Figure 9.2 presents the distributions of positive and negative emotional scores in bug-introducing and bug-fixing commit messages collected from the three projects. An ‘x’ mark in a box in the box-plot indicates the mean emotional scores over all the commits. As observed in Figure 9.2, for both bug-introducing and bug-fixing commit messages, emotional scores follow two similar patterns: (i) emotional scores of 75% commit messages remain within one to two in positively and negatively polarized commit messages and (ii) mean and median emotional scores in positively polarized messages are higher as opposed to mean and median emotional scores in negatively polarized messages.

We conduct a *MWW* test to verify statistical significance of difference between positive and negative emotional scores in bug-introducing commit messages. The test obtains p-value 2.57×10^{-11} where $p < \alpha$ and *Cliff's delta d* value -0.3032 where $|d| > 0.15$. Thus, those values indicate that the difference between positive and negative emotional scores in bug-fixing commit messages is significant.

Similarly, we conduct another *MWW* test between positive and negative emotional scores in bug-fixing commit messages to verify statistical significance of their difference. The test obtains p-value 2.2×10^{-16} where $p < \alpha$ and *Cliff's delta d* value -0.2246. Again, the obtained values indicate that the difference between positive and negative emotional scores in bug-fixing commit messages is significant.

Variation of a particular emotion across commit types: Next, we focus on emotion-wise differences between emotional scores in bug-introducing and bug-fixing commit messages. In other words, we want to verify statistical significances of difference between positive emotional scores found in bug-introducing and bug-fixing commit messages and difference between negative emotional scores found in bug-introducing and bug-fixing commit messages. From Figure 9.2, we see that the difference in mean positive emotional scores between bug-introducing and bug-fixing commit messages does not differ that much. Similar pattern can also be seen for difference in mean negative emotional scores between bug-introducing and bug-fixing commit messages.

To verify the statistical significance of our observations, we sequentially conduct two *MWW* tests. The first *MWW* test is conducted between positive emotional scores found in bug-introducing and bug-fixing commit messages. The test obtains p-value 0.076 where $p > \alpha$ indicates that the difference between positive emotional scores in bug-introducing and bug-fixing commit messages is not significant.

Similarly, we conduct another *MWW* test between negative emotional scores found in bug-introducing and bug-fixing commit messages, which obtains p-value 0.9561 where $p > \alpha$. Thus, the later also test indicates no significant difference between negative emotional scores in bug-introducing and bug-fixing commit messages.

Based on our observations and statistical tests, we derive the answer to the research question *RQ1* as follows:

Ans. to RQ1: *Both bug-introducing and bug-fixing commit messages have significantly higher positive emotional scores compared to negative emotional scores. However, neither positive nor negative emotional scores differ much between bug-introducing and bug-fixing commit messages.*

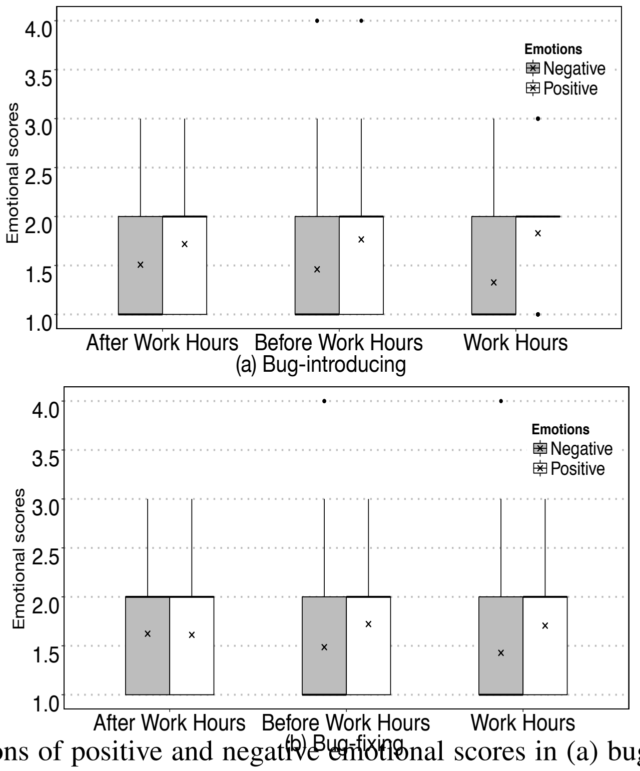


Figure 9.3: Distributions of positive and negative emotional scores in (a) bug-introducing and (b) bug-fixing commits messages

9.3.2 Hour-wise Emotional Variations

To study the relationship between developers emotions and times of a day when commit comments are posted, we divide the 24 hours of a day in three periods (a) 00 to 08 hours as *before working hours*, (b) 09 to 17 hours as regular *working hours* and (c) 18 to 23 hours as *after working hours*. Then, for each project, we organize the commit messages into three disjoint sets based on their timestamps of posting.

Variations of emotions in different commit types with respect to work hours: Figure 9.3(a) and 9.3(b) present the distributions of positive and negative emotional scores in bug-introducing and bug-fixing commit messages respectively posted in these three periods. We see from these figures that average positive emotional scores are always higher as opposed to average negative emotional scores except in *after work hours* for bug-fixing commit messages. Again, for both sentiments the median values in *after work hours* are equal only for bug-fixing commit messages, whereas for rest of the cases, the median values are always higher for positive sentiments. Another noticeable pattern can be observed for positive emotional scores in *work hours* for bug-introducing commit messages where the emotional scores are highly centered to value two.

Table 9.2: Results of *MWW* tests between positive and negative emotional scores of different commit types posted in different times of a day

Commit Types	**AWH**	**BWH**	**WH**
Bug-introducing	**0.0007, -0.2812**	**1.1e-10, 0.6943**	**2.2e-16, -0.1708**
Bug-fixing	0.4430, -0.3882	**6.8e-07, 0.6805**	**3.3e-08, -0.2941**
Here, AWH=After Work Hours, BWH=Before Work Hours, WH=Work Hours			

To verify the statistical significance of our observations, we conduct a series of *MWW* tests between positive and negative emotional scores in bug-introducing and bug-fixing commit messages respectively. The obtained *p*-values and their corresponding *d* values are presented (as a pair of *p*-value, *d*) in Table 9.2 where significant differences are marked bold. We see the differences are statistically significant in all cases except in *after work hours* for bug-fixing commit messages.

Variation of a particular emotion with respect to work hours: Next, we focus on emotion-wise difference between emotional scores of bug-introducing and bug-fixing commit messages posted in the three working periods. Figure 9.4(a) and 9.4(b) present the distributions of positive and negative emotional scores respectively in bug-introducing and bug-fixing commit messages in the three working periods. In those working periods, the averages of positive emotional scores are always higher in bug-fixing commit messages (see Figure 9.4(a)) whereas, interestingly, the averages of negative emotional scores are always higher in bug-introducing commit messages (see Figure 9.4(b)).

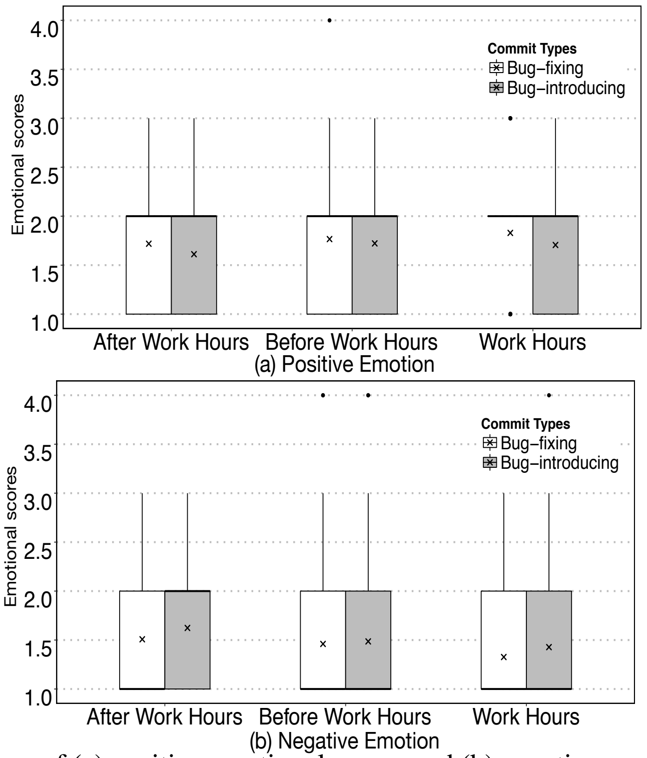


Figure 9.4: Distributions of (a) positive emotional scores and (b) negative emotional scores, in bug-introducing and bug-fixing commit messages

Table 9.3: Results of *MWW* tests of emotional scores between bug-introducing and bug-fixing commit messages posted in different times of a day

Emotion Types	**AWH**	**BWH**	**WH**
Positive	0.0866, 0.6640	0.2433, -0.2377	**0.0089, -0.1708**
Negative	0.9084, -0.4921	0.6677, 0.4075	0.9616, -0.6733

To verify the statistical significance of our observations, we again conduct a series of *MWW* tests between positive emotional scores in bug-introducing and bug-fixing commit messages and between negative emotional scores in bug-introducing and bug-fixing commit messages. The obtained *p*-values and their corresponding *d* values are presented in Table 9.3 where significant differences are marked bold. We see the only significant difference is between positive emotional scores in bug-introducing and bug-fixing commit messages posted during working hours.

Based on our observations and statistical tests, we now answer the research question *RQ2* as follows:

Ans. to RQ2: *Both bug-introducing and bug-fixing commit messages have significantly higher positive emotional scores compared to negative emotional scores in commits made in the three working hours, except for the after work hours of bug-fixing commit messages. During working hours, positive emotional scores in bug-introducing commits are significantly higher compared to the positive emotional scores in bug-fixing commits.*

9.4 Threats to Validity

Construct Validity: One may question the validity of our categorization of the developers' commits in different periods (Section 9.3.2), considering the possibility that the projects and developers may be physically located at different geographic locations and time-zones. However, we used the time zone information associated with the commit messages to convert all the timestamps to corresponding local times.

For the statistical tests of significance, we used the *Mann-Whitney-Wilcoxon (MWW)* test [132] and to compute effect size we used *Cliff's delta* [234]. The non-parametric *MWW* test along with non-parametric effect size *Cliff's delta* do not require the data to have normal distribution. Since the data in our work do not conform to normal distribution, this particular test suits well for our purpose. Moreover, the significance level α set to 0.05, which is a widely adopted value for this parameter that enables 95% confidence in the results of the *MWW* tests.

Internal Validity: The *internal validity* of our work depends on the accuracy of the tool's computation of emotional scores. `SentiStrength-SE` was reported to be effective in sentiment analysis [188] and suitable for extraction of emotions from commit comments. Nevertheless, the tool is not 100% accurate in determining emotional polarities of commit messages. We are aware of this threat and manually checked 50 comments and found their emotions were correctly identified by `SentiStrength-SE`.

One may question the accuracy of the approach to determine bug-introducing and bug-fixing commits. However, a manual checking found 96% accuracy of the approach in distinguishing bug-introducing and bug-fixing commits in the projects [232].

External Validity: The findings of this work are based on our study on more than 24,000 commit messages across three open-source projects. Generalizability of the results can be questioned due to small sized dataset.

Reliability: The methodology of this study including the procedure for data collection and analysis is well-documented in this chapter. The subject systems being open-source, are freely accessible while the tool `SentiStrength-SE` is also available online. Moreover, all the bug-introducing and bug-fixing commits are also available. Therefore, it should be possible to replicate the study.

9.5 Related Work

There are several studies exist in literature that compare emotional variations found in various textual artifacts related to software engineering. In such a study, Islam and Zibran [38] presented a quantitative empirical study of the emotional variations in different types of development activities and development periods, in addition to in-depth investigation of emotions' impacts on software artifacts. They found significantly higher positive emotion in bug-fixing commits compared to negative emotion, which is similar to our finding. From commit messages, Guzman et al. [34], Chowdhury and Hindle [36] and Sinha et al. [53] also identified emotions and group those in various dimensions. However, none of the above studies computed emotional scores in bug-introducing commits and compared against emotional scores in bug-fixing commit, that we have performed in our work.

Tourani et al. [43] extracted emotions from emails of both developers and system users. They observed the differences of emotional expressions between developers and users of a system. Garcia et al. [42] extracted developers' emotions from their email contents to analyze any relationships between developers' emotions and their activities in an open source software project. The studies of Tourani et al. [43] and Garcia et al. [42] differ with our work mainly in two ways (i) the source of their emotional content is different than ours and (ii) the objectives of those work are also orthogonal to ours.

Pletea et al. [41] mined developers' emotions commits and pull requests in `GitHub` projects. They analyzed emotional variations in discussions on different topics and reported to have found higher negative emotions in security-related discussions in comparison with other topics. While their objective, approach as well as source of emotional content and method of emotion extraction were different from our work, ours includes a deeper analysis based on emotional variations in bug-introducing and bug-fixing commit messages.

All the above mentioned studies used domain independent tools (e.g., *SentiStrength* and *NLTK*) to compute emotional scores in software engineering textual artifacts, however, for the first time, we have used a domain specific tool *SentiStrength-SE* for this study, which is one of the unique attributes of this work.

9.6 Summary

In this chapter, we have presented a quantitative empirical study on the emotional variations between bug-introducing and bug-fixing commit messages. We have studied more than 24,000 commit messages over three open-source projects.

In our study, we find that both bug-introducing and bug-fixing commit messages have overall statistically significantly higher positive emotional scores compared to negative emotional scores. We also observe similar findings while analyzing emotional scores in bug-introducing and bug-fixing commit messages with respect to three working periods. An exception is found in the later case where no significant difference found between positive and negative emotional score in bug-fixing commit messages posted during *after work hours*.

While comparing with respect to a particular sentiment (i.e., for positive or negative sentiment), we find no significant difference between overall emotional scores in bug-introducing and bug-fixing commit messages. The earlier finding also holds true while analyzing emotional scores in bug-introducing and bug-fixing commit messages with respect to three working periods with an exception where positive emotional scores are significantly higher in bug-fixing commits posted during *working hours*.

The findings from this work are validated in the light of statistical significance. Although more experiments can be conducted in large scale to verify or confirm the findings, the results from this study significantly advance our understanding of the impacts of emotions in software development activities.

Chapter 10

Roles of Affects in Software Engineering

In the previous chapter (Chapter 9), we have investigated the variation of sentiments in bug-fixing and bug-introducing commit messages to identify the potential role of sentimental variation in predicting software bugs. In this Chapter, we identify further roles of sentiments and emotions in software engineering activities. In Section 10.1, we identify the roles of sentiment in software design and quality. We present the roles and applications of human affects in software requirement and maintenance analysis in Section 10.2. Section 10.3 describes how developers' affects can be correlated with their performances. In Section 10.4, we identify how human affects can be utilized for effective communications in developers' social forums. Finally, the Section 10.5 summarizes the chapter.

10.1 Sentiments Analysis in Software Design and Quality

Lesiuk [62] recruited 56 software engineers to understand the effects of music listening on software design performance. Data was collected over a five-week period. The participants self-assessed affects and design performance. The results indicated that positive affects and self-assessed performance were lowest with no music, while time-on-task was longest when music was removed. Narrative responses revealed the value of music listening for positive mood change and enhanced perception on design while working.

Miller et al. [235] stated that a user's acceptance of products not only could be understood by their cognition but also by *sentiment*, while sentiment showed a deeper level of appealing. They proposed a flexible software modeling notation, called people-oriented software engineering (POSE) model, by adding sentimental goals to that for high-quality software design. They evaluated their models, using a case study and a user study, and found the importance of stakeholders' and users' sentiments towards software design as one influential dimension that could improve quality of the design.

Tourani and Adams [51] investigated the impact of various factors, including sentiments, on software quality (in terms of defect-prone commits). Sentiments of issue comments or review comments were computed, as well as a variety of other factors defined in their study. Using the factors as input variables, they built logistic regression models to study the impact of the characteristics (including sentiments) of issue and review discussions on the quality of a patch. However, they used a general purpose tool that had low accuracy in detecting sentiments.

10.2 Affective Analysis in Requirement Analysis and Software Maintenance

Maintenance requests (e.g., reporting of a bug, requests for feature, refactoring, and creation of test cases) that are submitted as issues in an issue tracking system (e.g., JIRA[1]) for a software system [236]. An issue gets closed when it is solved. However, a closed issue can be reopened when the patch or solution of the issue is deemed inappropriate, which, in turn, increases the amount of software maintenance tasks. Cheruvelil and Silva [237] studied a study to identify a correlation between sentiment and issue reopening. They collected 3,000 issue of eight projects from JIRA issue repository system. They used the tool `SentiStrength-SE` [109, 110] to detect sentiments of the issues comments. By conducting statistical analyses, they found evidence that negative sentiment in issue comments was correlated with issue reopening.

A common software maintenance task is to assign priority levels to bug reports submitted in an issue tracking systems. Some bugs are important and need to be fixed right away, whereas others are minor, and their fixes can be postponed until resources are available. However, manual prioritization is time consuming and cumbersome where automatic approaches can be an alternative to make the software maintenance job easy. Umer at al. [238] developed an *emotion-based automatic approach* to predict the priority for a bug report. They collected 80,000 bug reports of four projects of Eclipse from Bugzilla[2]. For each report, they computed (i) emotion-value using `SentiWordNet` [159] dictionary, and (ii) term frequency of feature words. Those computed emotion-values and terms' frequencies are used as features in Support Vector Machine [214] to develop the predictor. After training and testing of the emotion-based predictor, it was found that it outperformed a state-of-the-art technique in predicting priorities of bug reports.

Umer et al. [239] proposed a sentiment based approach to predict how likely enhancement reports would be approved or rejected so that developers could first handle likely-to-be-approved requests. First, they preprocessed enhancement reports using natural language preprocessing techniques. Second, they identified the words having positive and negative sentiments in the summary attribute of the enhancements reports and calculated the sentiment of each enhancement report. Finally, with the history data of real software application, they trained a machine learning based classifier to predict whether a given enhancement report would be approved. The proposed approach was evaluated with the history data from real software applications. The cross-application validation suggested that the proposed approach outperformed the state-of-the-art. The evaluation results suggested that the proposed approach increased the accuracy from 70.94% to 77.90% and improved the F-measure significantly from 48.50% to 74.53%.

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

Yang et al. [137] successfully predicted severities of bug reports using sentiments expressed in those reports. To predict bug severity, they developed *EWD-Multinomial* algorithm, which is a variant of Naive Bayes Multinomial (NBM) [125] classifier. In a separate study, Yang et al. [240] also developed a emotion-based novel approach to predict severity of reported bugs. They computed the probability distributions of emotional words in bug reports and used that distributions to train an NBM classifier to predict severities of the bug reports. 10-fold cross-validation was used to train and test the classifier using 23,929 bug reports collected from five open-source projects. Test results showed that their emotion-based approach outperformed other state-of-the-art techniques.

Colomo-Palacios et al. [107] showed how emotions can indicate acceptance of requirements. They explicitly ask stakeholders about their feelings regarding particular requirements. Their responses are recorded in a grid which represents in the X axis the arousal caused by the requirement and in the Y axis the pleasure. Comparing the responses with the evolution of the requirements throughout the requirement engineering process iterations, they conclude that high levels of pleasure and low levels of arousal seem to indicate accepted requirements.

Colomo-Palacios et al. [28] aimed to integrate the developers' and the system users' emotions into the process of requirements engineering. They conducted a study on two software projects and 11 individuals. In total, 65 user requirements were produced between the two projects, which lasted six months and seven months, respectively. Each requirement faced tens of revision. Each participant rated the affects associated to each requirement version. The results showed that the affects related to pleasantness associated to the final requirements are higher than for non-final requirements, while the affects related to mental activation (arousal) for the final requirements are lower than for non-final requirements.

Miller at al. [235] argued that users' emotional goals/needs should be considered as requirements to develop a system that could ensure users' satisfaction. They designed and implemented a new prototype of a wellbeing-check emergency system for older peoples that considered emotional goals of it's users. To evaluate the user satisfactions in using the emergency system, they placed it into the homes of nine older people over a period of approximately two weeks each. Interviews were conducted both before and after the deployment period to evaluate users' satisfaction. The data gathered by interviews demonstrated that considering users' emotions in a system led to improved users' satisfaction.

Williams and Mahmoud [241] performed a systematic in-depth analysis in the domain of anonymous social networking apps (e.g., Whisper[3] and Firechat[4]) and proposed a model to depict the interrelationships between apps users' sentimentally polarized (i.e., positive and negative) concerns and the core features of the apps in the domain. They argued such a model (that included senti-

[3]`http://whisper.sh/`

[4]`https://www.opengarden.com/firechat.html`

ment analysis) could be utilized to identify urgent users' concerns and help app developers to devise sustainable release engineering strategies.

Carreno and Winbladh [242] adapted the Aspect and Sentiment Unification Model (ASUM) [243] that incorporated both topic modeling and sentiment analysis techniques to identify requirement change requests mentioned in user comments on mobile applications. They applied the ASUM model on three different sets of user comments on mobile applications and examined the effectiveness of the model in identifying requirement change requests. The experimental results showed that the model clearly generated a good representation of topics that could be relevant with regard to requirements changes.

Guzman and Maalej [182] also combined sentiment analysis and topic modeling to develop an automated approach to systematically analyze user opinions on various applications' features. They used natural language processing techniques to identify fine-grained app features in the reviews. Then, they used the tool `SentiStrength` [45] to compute users' sentiments from users reviews about the identified features. Finally, they used topic modeling techniques to group fine-grained features into more meaningful high-level features. They evaluated their approach with seven apps from the Apple App Store and Google Play Store and compared its results with a manually, peer-conducted analysis of the reviews. Their approach could help app developers to systematically analyze user opinions about single features and filter irrelevant reviews.

One of the major risks associated with software development is related to the phenomenon of over-requirement. The over-requirement phenomenon is manifested when a product or a service is specified beyond the actual needs (i.e., nice-to-have a product or a service but not necessary) of the customer or the market. Shmueli et al. [244] conducted a study to show that over-requirement was due partially to the emotional involvement of the developers with the features they specified. This emotional involvement seemed to be associated with three effects that had been demonstrated by behavioral economists: (i) Endowment effect, i.e., the tendency of people to overvalue their possessions [245], (ii) IKEA effect, i.e., the tendency of people to overvalue their self-constructed products [246], (iii) I-designed-it-myself effect, i.e., the tendency of people to overvalue their self-designed products [247]. To explore these behavioral effects and the interactions among them in the context of software development, they conducted an experiment in which over 200 participants were asked to specify a nice-to-have software (but unnecessary) feature. Their results confirmed the existence of these behavioral effects in software development and their influence on over-requirement.

Fu et al. [248] proposes `WisCom`, a system that could analyze tens of millions user ratings at three different levels (i.e., micro, meso and macro) by leveraging sentiment analysis expressed in apps' comments. In the micro level analysis, they performed word-level analysis of review comments to understand the impact of each word on users' actual sentiments. This analysis helped them to identify the vocabulary users used to praise or criticize apps. By applying a regularized regression model, they developed a predictor to predict the rating score based on the comments users posted.

The predictor helped them to detect inconsistent ratings that did not match the actual texts of the comments. They found this type of inconsistency in roughly 0.9% of the user reviews they collected. In meso level analysis, they aggregated comments of individual apps and used text analysis (e.g., topic modeling [249]) to further study why users disliked apps. They further extend their analysis to the scope of whole marketplace in the macro level analysis. They aimed at understanding the general user preferences and concerns over different types of apps and providing guidelines to developers or even market operators.

To identify evolutionary requirements for software systems, Jiang et al. [250] proposed a systematic approach based on users' sentimental opinions expressed in online reviews. At first, they automatically extracted positive and negative opinions expressed in the reviews about common software features. Then, they grouped similar opinion expressions about software features. Next, they produced the structured feedback classified by software features, including several corresponding sentences. Afterwards, they measured users' satisfaction scores about a software feature based on the number and intensity of positive and negative opinions of the feature. Finally, based on the measured satisfaction scores of features, they manually generated the document of evolutionary requirements for each software feature. To evaluate the usefulness of the evolutionary requirements document generated by their approach, they conducted a human subjective study with 50 developers. The study found that the generated document could help developers understand why and what to evolve for future software releases.

Zhao and Zhao [251] proposed a framework that leveraged sentiment analysis to analyze users' reviews to automatically predict requirement evolution of products' features. The proposed a framework combined a supervised deep learning neural network [] with an unsupervised hierarchical topic model to analyze user reviews automatically for product feature requirements evolution prediction. The approach could discover hierarchical product feature requirements from the hierarchical topic model and identify their sentiment by the Long Short-term Memory [252] with word embedding [253], which not only models hierarchical product requirement features from general to specific, but also identifies sentiment orientation to better correspond to the different hierarchies of product features. The evaluation and experimental results show that the proposed approach was effective and feasible.

Zhou et al. [254] devised a technique to augment a software product line planning [255] technique by leveraging sentiment analysis of user-generated online product reviews. The technique consisted of three steps: (i) hybrid sentiment classification of product reviews with lexicons and a machine learning technique, named rough set, (ii) features' extraction from product reviews using association rule mining and user-defined features, and mapping features and sentiments to determine customers' preferences, and (iii) supporting a software product line planning technique using the customers' preferences. They demonstrated the feasibility and potential of the proposed technique via an application case.

Panichella et al. [168] developed a technique that used textual analysis (TA), natural language parsing (NLP), and sentiment analysis (SA) to detect and classify app users' intensions (e.g., feature request, problem/bug discovery) expressed in users' review comments. They applied TA to preprocess texts and compute word frequency of each word in processed texts. They applied NLP technique to manually inspect 500 reviews to identify 246 recurrent linguistic patterns[5]. For each identified linguistic pattern they formalized and implemented an NLP heuristic to automatically recognize it from review comments. To detect sentiment of each review comment, they trained Naive Bayes classifier using word frequency as a feature. They used words' frequencies, linguistic patterns, and sentiments as features in different machine learning techniques, namely, the standard probabilistic Naive Bayes classifier, Logistic Regression, Support Vector Machines, J48, and the alternating decision tree to implement classifiers to detect app users' intentions. To evaluate the classifiers, they created a ground truth dataset of 1,421 sentences that were labeled according to their intentions by two raters. Then, they ran the machine learning classifiers using different combinations of the features on the ground truth dataset and measured the performances of the classifiers using precision, recall, and F-measure. The results show that the combination of all features achieved the best results with the J48 algorithm, among all possible feature inputs and classifiers with 75% precision and 74% recall. Later, Panichella et al. [256] developed and published a tool named `ARdoc` using the described technique [168]. However, they used Stanford CoreNLP [146] instead of Naive Bayes classifier to determine sentiments of reviews to develop `ARdoc`.

Sorbo et al. [257, 258] developed a tool named `SURF` by using the tool `ARdoc`. `SURF` could automatically (i) extract the topics (e.g., security, pricing, and content) expressed in reviews, (ii) classify the intention (by employing `ARdoc`) of the writers, to suggest the specific kinds of maintenance tasks developers have to accomplish, and (iii) group together sentences covering the same topic. They conducted an empirical study involving 12 developers/engineers from different companies in Switzerland, Italy and the Netherlands and 11 developers/engineers from the development team of Sony Mobile in Japan, to investigate the practical usefulness of summaries generated by SURF in the developers' "working context". The empirical study found that `SURF` was able to summarize thousands of app reviews in form of an interactive, structured and condensed agenda of recommended software changes.

Palomba et al. [259] introduced a novel approach that analyzes the structure, semantics, and sentiments of sentences contained in user reviews to extract useful (user) feedback from maintenance perspectives and recommend to developers changes to software artifacts. It relied on natural language processing and clustering algorithms to group user reviews around similar user needs and suggestions for change. Then, it involved textual based heuristics to determine the code artifacts that needed to be maintained according to the recommended software changes. The quantitative and qualitative studies carried out on 44683 user reviews of 10 open source mobile apps and their

[5]`http://www.ifi.uzh.ch/seal/people/panichella/Appendix.pdf`

original developers showed a high accuracy of the approach in (i) clustering similar user change requests and (ii) identifying the code components impacted by the suggested changes. Moreover, the obtained results show that the approach was more accurate than a baseline approach for linking user feedback clusters to the source code in terms of both precision (+47%) and recall (+38%).

Gu and Kim [260] proposed a Software User Review Miner (SUR-Miner), a framework that could summarize users' sentiments and opinions toward corresponding software aspects. Instead of treating reviews as bag-of-words, SUR-Miner made full use of the monotonous structure and semantics of software user reviews, and directly parsed aspect- opinion pairs from review sentences based on pre-defined sentence patterns. It then analyzed sentiments for each review sentence and associate sentiments with aspect-opinion pairs in the same sentence. Finally, it summarized software aspects by clustering aspect-opinion pairs with the same aspects. They empirically evaluated the performance of SUR-Miner on recent user reviews of 17 Android apps such as Swiftkey, Camera360, WeChat and Templerun2. Results showed that the SUR-Miner produces reliable summaries, with average F1-scores of 75%, 85%, and 80% for review classification, aspect-opinion extraction and sentiment analysis, respectively. The final aspects from SUR-Miner are significantly more accurate and clearer than state-of-the-art techniques, with an F1-score of 81%.

Maalej and Nabil [261] applied several machine learning algorithms (e.g., Naive Bayes, Decision Tree, and MaxEnt) to classify app reviews into four types: bug reports, feature requests, user experiences, and ratings (that included great, good, nice, very, cool, love, hate, bad, worst). In this work, first, they collected real reviews from app stores and extracted their metadata (e.g., the star rating and text length). Second, they created a ground truth dataset by selecting a representative 4,400 samples of the collected reviews and labeled them according to their review types by manually analyzed their content. Third, they implemented different classifiers using review metadata (e.g., star rating, length, and tense of a comment), bag-of-words of preprocessed comments, and sentiment scores (computed using `SentiStrength`) as the features of the machine learning algorithms. Fourth, they split the ground truth dataset at a ratio of 70:30 where 70% of the ground truth dataset were used to train the classifiers, and 30% for testing. Fifth, they used the training set to train the classifiers and test set to test the accuracies of the classifiers. Finally, they ran a series of experiments using the classifiers on the test set to evaluate the performances of the classifiers in terms of precision, recall, F-measure. The experimental results showed that when metadata was combined with text processing and sentiment analysis, the classification precision was increased.

Williams and Mahmoud [262] used textual analysis and sentiment detection technique to accurately capture and categorize the various types of actionable software maintenance requests (e.g., bug reports and user requirements) existed in Twitter messages. First, they collected 4,000 unique Twitter messages related to 10 software systems and manually classified them according to their maintenance requests. Second, they applied textual analysis to preprocess texts that included stemming and stop-word removal to reduce the number of features (i.e., words). Third, they computed sentiment scores of messages using SentiStrength. Fourth, the existing words, i.e., bag-of-words in messages (remained after preprocessing) and computed sentiment scores of those messages were used as features in two text classification machine learning classifiers, namely Naive Bayes and Support Vector Machines. They used 10-fold cross validation to train and test the classifiers. The test results showed that both Naive Bayes and Support Vector Machines were able to achieve competitive results to capture and categorize the various types of actionable software maintenance requests.

Guzman et al. [263] conducted an experiment to investigates the performance of individual machine learning algorithms, namely Naive Bayes, Support Vector Machine, Logistic Regression and Neural Networks and their ensembles for automatically classifying the app reviews in different categories (e.g., bug report, praise, and complaint). They converted review texts into a vector space model and used TF-IDF as a weighting scheme and used additional features, such as review rating, number of words in the review, ratio of positive sentiment words, and ratio of negative sentiment words. They evaluated the performance of the machine learning techniques on 4550 reviews that were systematically labeled using content analysis methods. Overall, the ensembles had a better performance than the individual classifiers, with an average precision of 74% and 59% recall.

Guzman at al. [264] presented ALERTme, an approach to automatically classify, group and rank tweets about software applications. They applied machine learning techniques for automatically classifying tweets requesting improvements, topic modeling for grouping semantically related tweets and a weighted function for ranking tweets according to specific attributes, such as content category, sentiment (detected by SentiStrength) and number of retweets. They ran ALERTme on 68,108 collected tweets from three software applications and compared its results against software practitioners' judgements. The experimental results showed that ALERTme was an effective approach for filtering, summarizing and ranking tweets about software applications. ALERTme enabled the exploitation of Twitter as a feedback channel for information relevant to software evolution, including end-user requirements.

In another study, Jha and Mahmoud [265] again used textual analysis and sentiment detection technique to classify non-functional requirements (NFR), namely dependability, performance, supportability, and usability expressed in the review comments of iOS mobile applications. First, they collected 6,000 reviews and manually annotated those reviews according to the expressed NFRs by human raters. Second, they preprocessed texts to perform word stemming and removing stop-words to reduce the number of features (i.e., words). Third, they computed sentiment scores of messages using the the dictionary VADER - Valence Aware Dictionary and sEntiment Reasoner [120]. They also identified the domains (e.g., music, game, and education) of the apps. Fourth, the existing words, i.e., bag-of-words in messages (remained after preprocessing), and computed sentiment scores of those messages, and domains of the apps were used as features in two text classification machine learning classifiers, namely Naive Bayes and Support Vector Machines. Fifth, they split the ground truth dataset at a ratio of 70:30 where 70% of the ground truth dataset were used to train the classifiers, and 30% for testing. Finally, they used the training set to train the classifiers and test set to test the accuracies of the classifiers. The test results showed that classification features, such as the sentiment score of the review led to slight improvement to the accuracies of the classifiers in classifying the NFRs.

Nayebi et al. [266] applied three machine learning algorithms to predict marketability and success of a new release of an app (in source code repository, e.g., GitHub[6]). The term “Marketability" referred to the question if a new release should be marketed or not. The success of a marketed app was measured using sentiment analysis of users’ reviews of the app. They gathered a pool of 11,514 releases over 917 apps. Among them, 7,435 releases were published and 4,079 releases were not published. Among the 7,435 marketed releases, they identified 3,734 releases as successful and 3,701 releases as unsuccessful. Then, they extracted 12 release attributes (e.g., number of changed files and number of open issues) and three app attributes (e.g., sentiment) from each app. The extracted attributes were used as features in Decision Tree, Support Vector Machine, and Random Forest machine learning prediction models. They evaluated the accuracy of the prediction models by using both 10-fold and Leave-One-Out cross-validation (LOOCV). The performances of the three prediction models were about the same while Random Forest had a slightly better F1 scores. Besides, app attributes in conjunction with release attributes contributed in better recall with precision being about the same.

Uddin and Khomh [267, 208] built a suite of techniques to automatically mine and categorize opinions about APIs from forum posts. First, they detected opinionated sentences in the forum posts. In the study, s sentence was opinionated if it consisted of positive or negative sentiment. They developed a technique named `OpinerDSOSenti` to detect sentiment in the sentences. Second, they associated the opinionated sentences to API mentions. Third, they detected API aspects (e.g., performance, usability) in the sentences. They developed and deployed a tool called `Opiner`, supporting the above techniques. `Opiner` is available online as a search engine, where developers can search for APIs by their names to see all the aggregated opinions about the APIs that are automatically mined and summarized from developer forums.

Lin et al. [268] proposed an approach named `POME` that leveraged natural language parsing and pattern-matching to classify Stack Overflow sentences referring to APIs according to seven aspects (e.g., performance, usability), and to determine their polarity (positive vs negative). By combining aspects and sentiments in sentences, the approach identified opinions related to APIs. Total 157 patterns were inferred by manually analyzing 4,346 sentences collected from Stack Overflow linked to a total of 30 APIs. They evaluated POME by (i) comparing the pattern-matching approach with machine learners leveraging the patterns themselves as well as n-grams extracted from Stack Overflow posts; (ii) assessing the ability of POME to detect the polarity of sentences, as compared to sentiment-analysis tools. (iii) comparing `POME` with the state-of-the-art Stack Overflow opinion

[6]`https://github.com`

mining approach, `Opiner`, through a study involving 24 human evaluators. The evaluation showed that `POME` exhibited a higher precision than a state-of-the-art technique (`Opiner`), in terms of both opinion aspect identification and polarity assessment.

10.3 Correlational Analysis between Developers' Affects and Their Performances

Ortu et al. [40] conducted a study to identify a correlation between developers' affectiveness (e.g., sentiment and politeness) and time needed by developers to fix issues (e.g., bug) reported in an Issue Tracking System (ITS). First, they collected 560K issue comments posted between 2002 to December 2013 in the ITS Jira [7]. Then, they extracted eight control metrics (e.g., issue priority, issue type) and 12 affective metrics (e.g., average sentiment of an issue, sentiment in the title of a sentiment). Next, they used a hierarchical modeling approach where one metric at a time was added to build a Logistic Regression model. The model was compared using an ANOVA test[8] to the previous model (without that metric) to check whether the addition of the metric led to a statistically significant improvement (i.e., $p\text{-value} < 0.01$ for the metric) of the model. They found six control metrics and nine affective metrics were significant. Those significant metrics were used to build the final Logistic Regression model. The evaluation results showed that the Logistic Regression model achieved a precision of 67% and recall of 67.1% against respectively 31.9% and 56% for the ZeroR model. They also built two variants of the Logistic Regression model by including and excluding the affective metrics and testes their performances in predicting issues' fixing times. The test results also confirmed that addition of the affective metrics to the model increased precision and recall values.

While the above study explored the correlation between issue fix time and affective metrics (e.g., emotions and sentiment), Mäntylä et al. [39] investigated the role of Valence, Arousal, and Dominance (VAD) metrics to correlate with issue fix time. They used the issue reports collected by Ortu et al. [40] to conduct the investigation. For each report, they extracted title, description and comments, then calculated the VAD metrics using the a lexica [160] contained 13,915 English words with VAD scores for Valence, Arousal and Dominance. Thet also computed the control and affective metrics used by Ortu et al. [40]. Then, they built a logistic regression model to investigate which variables were associated the most with issue resolution time. They used logistic regression to build the model hierarchically to predict output variable (i.e., issue fix time) into a binary variable: “Short" (fix time lower than median fix time) and “Long" (fix time larger than or equal to median fix time). First, they built a model using the control metrics used by Ortu et al. [40], then a second model was built using both the control and affective metrics of Ortu et al. [40]. Finally, a model with all control, affective, and VAD metrics was built. To evaluate precision and recall, they used

[7] `https://www.atlassian.com/software/jira`
[8] `https://en.wikipedia.org/wiki/Analysis_of_variance`

10-fold cross-validation. The evaluation results showed that F-score of the final model significantly increased (from 71.7% to 75%) after adding VAD metrics with control and affective metrics. They also found that across VAD metrics, valence of all comments and the valence of the issue title had the largest impact on issue fix time.

Graziotin et al. [60] conducted an investigation on the correlation of affective states of software developers and their self-assessed productivity. They observed eight developers working on their individual projects. Their affective states (i.e., valence, arousal, and dominance) and their self-assessed productivity were measured on intervals of ten minutes using Self- Assessment Manikin (SAM) [269]. A linear mixed-effects model [60] was used to estimate the value of the correlation of the affective states of valence, arousal, and dominance, and the productivity of developers. The model's values revealed that there were positive correlations between two affective states dimensions (valence, dominance) and the self-assessed productivity of software developers.

Graziotin et al. [270] echoed the call for research on alternative factors influencing the performance of software developers. They conducted a study with 42 computer science students to investigate the relationship between the affects and creative and analytical performance of software developers. The participants performed two tasks coming from psychology research. The first task was related to creative performance, the other was related to analytic performance and resembled algorithm design and execution. The participants? pre- existing affects were measured before each task. The analysis of the data showed empirical support for the claim that happy developers are indeed better problem solvers in terms of their analytical abilities. The study raised the need for studying the human factors of software engineering by employing a multidisciplinary viewpoint.

To explore the impacts of sentiments on the debugging performance of software developers, Khan et al. [61] used high-arousal-invoking and low-arousal-invoking movie clips to trigger different levels of sentiments in developers before having them perform some debugging tasks. In the experiment, 72 programmers watched selected short movie clips that provoked positive and negative moods. Afterward, they completed a debugging test. Results showed the video clips that swung programmers sentiments/moods had a significant effect on programmers' debugging performances. However, they did not employ any measurement to extract and quantify the developer's sentimental states and relied on the assumption that watching those movie clips would induce different levels of sentiments in the developers.

Garcia et al. [42] analyzed the relation between the emotions and the activity of contributors in the Open Source Software project Gentoo. They built datasets from the project's bug tracking platform Bugzilla, to quantify the activity of contributors, and its mail archives, to quantify the emotions of contributors by means of sentiment analysis. The Gentoo project is known for a period of centralization within its bug triaging community. This was followed by considerable changes in community organization and performance after the sudden retirement of the central contributor. They analyzed how this event correlated with the negative emotions, both in bilateral email discussions with the central contributor, and at the level of the whole community of contributors. They then extended the study to consider the activity patterns on Gentoo contributors in general. The found that contributors were more likely to become inactive when they express strong positive or negative emotions in the bug tracker, or when they deviated from the expected value of emotions in the mailing list. They used these insights to develop a Bayesian classifier that detected the risk of contributors leaving the project. Their analysis opened new perspectives for measuring online contributor motivation by means of sentiment analysis and for real-time predictions of contributor turnover in Open Source Software projects.

Wrobel [31] also found correlations between developers' sentiments and their productivities. He pointed out that sentimental state *frustrated* was the riskiest emotional state in terms of developers' productivity. Graziotin et al. [60] conducted a study on how sentiments (including dominance, valence, arousal dimensions) impacted the productivity of developers. They selected eight participants (four students and four professional developers) and used questionnaires for measuring the affects of participants and their productivities. Their experimental results showed that the happiness of a developer (i.e., positive sentiment) was positively correlated with self-assessed productivity. In both studies, the sentimental states of the participants were identified manually by questionnaires and/or interviews.

Results, obtained from a four years project conducted by Ortu et al. [271], showed that positive sentiments and good manners positively impacted both productivity and wellness of developers.

10.4 Sentiment Analysis in Software Social Forums

Calefato et al. [134] argued that the emotional style (with other factors, e.g., presentation quality, user's reputation) could increase the chance of getting their answer accepted in Stack Overflow (SO) [9] posts. They started an investigation along that direction by collecting a dataset contained 348,618 answers from a SO data dump that was updated on September 2014. They computed five factors: acceptance vote, presentation quality, sentiment, temporal, and social metrics of each answer. They used `SentiStrength-SE` to compute sentiment of the answers of SO posts. They used acceptance vote as the dependent variable and the remaining four factors as independent variables in a logistic regression classifier. They split the dataset at a ratio of 70:30 where 70% of the dataset were used to train the classifier, and 30% for testing. They assessed the classifier's quality in term of Receiver Operating characteristic Area Under Curve and performed the experiment in an ablation test setting by removing one of the factors at a time, while retaining the others. They found evidence that factors related to information presentation, time, and sentiment had an impact on the success of answers got accepted.

[9] `https://stackoverflow.com`

SO has become a popular place for discussing different types of API issues, such as incomplete or erroneous documentation, poor performance, and backward incompatibility. Ahasanuzzaman et al. [272] proposed a supervised learning approach using Conditional Random Fields (CRFs) [273] to classify sentences in SO posts either as issue or non-issue. They considered a SO post was related to an issue if that contained a link of a particular issue tracker address in the post's issue description. They collected 2,500 SO posts that contained links of issue trackers and conducted a manual study to ensure that all these posts were related to various issues. They used bag-of-words, parts-of-speech, sentiment, and positions of API issue-related sentences in posts as features to train a CRFs based classifier. SentiWordNet 3.0 [158] was used to detect sentiments of posts. Although, they did not report the accuracy of the classifier, the developed CRFs classifier was successfully used in another classifier to categorize issues in different types.

Rahman et al. [154] proposed a mining technique that mined insightful comments from Stack Overflow for a given code segment, where the comments reveal identified issues, deficiencies and scopes for further improvement in the code. Their approach mined the comments by analyzing five aspects of comments– popularity, relevance, comment rank, word count and sentiment expressed in the text. Experiments with 292 Stack Overflow code segments and 5,039 discussion comments showed that their approach could extract the insightful comments with a promising recall of 85.42% and a MRR of 0.44 on average. A user study with professional developers and 85 code segments also showed that about 80% of the recommended comments were found accurate, precise, concise and useful by the participants.

Jiarpakdee et al. [136] investigated the impact of affective features (e.g., sentiment and politeness) on quality of a question asked/posted in SO. In the study, a question was considered to have high-quality if that question got an accepted answer. To conduct their investigation, first, they collected 38,231 questions from the SO dataset provided for MSR 2015 Challenge[10]. Second, from the questions, they extracted 31 features categorized into three classes: text-based (e.g., number of sentences in questions), community-based (e.g., reputation of asker), and affective. Third, they implemented models by using different combinations of the features in a random forest algorithm [274]. To train and test the models, they applied 10-fold cross validation by repeating that 100 times. Finally, after testing, they applied Scott-Knott test [275] to reveal what features were influential in their models. The test results showed that community-based and affective features play an important role in the question quality identification.

Mondal et al [276] proposed a sentiment metric based machine learning model for identifying low-quality and high-quality questions in asked SO. They defined a question's quality based on up-vote and down-vote of the question. They considered that a question was of low quality if that got down-votes by in SO and vice versa. They collected 38,920 SO questions submitted between January, 2014 and January, 2015 and identified qualities of those questions. For question quality

[10]`http://2015.msrconf.org/challenge.php`

prediction, they computed four features, i.e., (i) TF-IDF [277] of key/indicator terms in the question texts, (ii) negative sentence count, (iii) positive sentence count and (iv) neutral sentence count from a question. Sentiments of sentences were detected by `Sentiment140`API [11]. These features were used in Multilayer Perceptron [278] and Support Vector Machine [279] algorithms to build the classifier models to classify/predict qualities of questions. They used 10-fold cross-validation for training and testing the performance of the developed models. For evaluation, they used precision and recall metrics. Evaluation results suggested about 70% precision and about 74% recall for their model that clearly revealed the impact of human emotions upon the quality of questions.

Using sentiment analysis of SO posts, researchers not only tried to identify quality of a question but also quality of source code. Serva et al. [280] developed an automatic sentiment analysis-based technique for mining poorly written code examples from developer question and answer forums along with a technique to automatically.

10.5 Summary

In this chapter, we have identified the roles and applications of affective analysis in various software engineering activities. We have found that affective analysis has been widely used in different software engineering activities that include software requirement and maintenance analysis, software design and quality, improving developers' performance, various correlational analysis to identify potential use of human affects, and devising effective communication mechanisms in developers' social forums.

[11]`http://help.sentiment140.com`

Chapter 11

Code Smells in Categories of Clones and Non-Cloned Code

From Chapter 4 to Chapter 9, we presented our studies to address the sub-problem I (i.e., detecting developers' sentiments/emotions) and sub-problem II (i.e., understanding developers' sentiments). The studies that we conducted to address subproblem-III (i.e., understanding impacts of developers copy-paste actions) are presented in Chapter 11, Chapter 12, and Chapter 13. In Chapter 11, we describe a comparative study on different types of clones (i.e., copy-pasted code) and non-cloned code on the basis of their code-smells, which may lead to software defects and other issues in future.

The rest of the chapter is organized as follows. In Section 11.1, we introduce our motivation and context of the work. We define the terminology and metrics used in this study in Section 11.2. In Section 11.3, we provide the details of the experimental setup and procedure of our empirical study. Section 11.4 presents our analysis and the findings of this study. In Section 11.5, we discuss the possible threats to the validity of our study. Section 11.6 includes related work, and Section 11.7 concludes the chapter.

11.1 Introduction

Source code reuse by copy-paste is a common practice that software developers adopt to increase productivity. Such a reuse mechanism typically results in duplicate or very similar code fragments commonly known as *code clones*. Aside from such deliberate cloning, unintentional clones are also created for various reasons under diverse circumstances [66, 67]. Software systems typically have 9%-17% [68] cloned code, and the proportion is sometimes found to be even 50% [69] or higher [70].

Despite the few benefits [71] of cloning, code clones are detrimental in most cases [72, 71, 73]. Code clone is a notorious code smell (i.e., a symptom indicating source of future problems) [74], that cause serious problems such as *reduced code quality*, *code inflation*, *program faults*, *security vulnerabilities*, and *bugs propagation* [72, 75]. Clones are thus a major contributor to the high maintenance cost for software systems, and as much as 80% of software costs are spent on maintenance [76]. Therefore, it is necessary to keep the number of clones at the minimum and to remove them from source code by refactoring. However, not all the clones in a software system are harmful [71], neither it is feasible to remove all the clones in source code by refactoring [77, 75]. Therefore, we must distinguish the context and characteristics of clones, which make them malign as opposed to the benign clones.

Towards this goal, several studies have been performed in the past to examine or exploit comparative stability of clones as opposed to non-cloned code [78, 79, 80, 81, 82], relationships of clones with bug-fixing changes [83, 84, 85, 72, 86, 87, 25, 88, 89], and the impacts of clones on program's changeability [90, 73, 91]. Note that, code smells are symptoms of poor coding patterns and are probable sources of future serious problems. While code clone itself is a notorious code smell [74] other fine grained code smells, defects and vulnerabilities often hide inside cloned code. Thus, such "*smelly code clones*" can be regarded as *bug-prone* clones, which are likely to cause serious defects, and the reuse by copy-pasting of such a bug-prone piece of code causes multiplication of bug-proneness elsewhere in the software system.

Earlier attempts [85, 86, 87, 89] to determine bug-proneness of code clones relied on long-term history of bug fixing changes preserved in version control system. While such studies make important contributions, their approaches do not fit well for proactive clone management, especially at the early stages of software development process where significantly long history of bug-fixing changes are not available. Therefore, to determine bug-proneness of code clones, we choose to apply static source code analysis techniques that do not require any bug-fixing history.

This chapter presents an empirical study on the relationships of program vulnerabilities with code clones. Here *'vulnerability'* refers to problems in the source code identified based on bad coding patterns [74], which lead to bugs, security holes, performance issues, design flaws, and other difficulties. A list of such vulnerabilities are presented in Table 11.1. A particular piece of code is considered vulnerable if it contains code smells or bad coding patterns, and the severity of the vulnerability is dictated by the severity of existing code smells. In this work, we address the following three research questions.

RQ1: *Are code clones more vulnerable than non-cloned code or vice versa*? — Rahman et al. [25] reported that the great majority of software defects are *not* significantly associated with clones, while Juergens et al. [72] claimed otherwise.

RQ2: *Are clones of a certain category relatively more vulnerable than others?* — If a certain type of clones are found to be more vulnerable, those clones can be high-priority candidates for removal or careful maintenance.

RQ3: *Is there a particular set of vulnerabilities that appear more frequently in cloned code as opposed to non-cloned code?* — If such a set of vulnerabilities can be identified, the findings will help software developers staying cautious of such vulnerabilities while cloning source code. In addition, those particular set of vulnerabilities can be avoided or removed by the use of clone refactoring.

To answer the aforementioned research questions, we conduct a quantitative empirical study over 97 open-source software systems drawn from diverse application domains. Using a wide range of metrics and characterization criteria, we carry out an in-depth analysis on the source code of the systems with respect to different categories of code clones, non-cloned code, and a diverse set of vulnerabilities.

11.2 Terminology and Metrics

In this section, we describe and define the terminologies and metrics used in our work.

11.2.1 Characterizing Terminologies

Our study includes clones at the granularity of syntactic blocks at different levels of similarities.
Type-1 Clones: Identical pieces of source code with or without variations in whitespaces (i.e., layout) and comments are called *Type-1* clones [67].
Type-2 Clones: *Type-2* clones are syntactically identical code fragments with variations in the names of identifiers, literals, types, layout and comments [67].
Type-3 Clones: Code fragments, which exhibit similarities as of *Type-2* clones and also allow further differences such as additions, deletions or modifications of statements are known as *Type-3* clones [67].

Notice that by the definitions above, *Type-2* clones include *Type-1* while *Type-3* clones include both *Type-1* and *Type-2*. Let, T_1, T_2, and T_3 respectively denote the sets of *Type-1*, *Type-2*, and *Type-3* clones in a software system. Mathematically, $T_1 \subseteq T_2 \subseteq T_3$. Thus, we further define two subsets of *Type-2* and *Type-3* clones as follows.
Pure Type-2 Clones: A set of pure *Type-2* clones include only those *Type-2* clones that do not exhibit *Type-1* similarity. Mathematically, $T_2^p = T_2 - T_1$, where T_2^p denotes the set of pure *Type-2* clones.
Pure Type-3 Clones: A set of pure *Type-3* clones include only those *Type-3* clones, which do not exhibit similarities at the levels of *Type-1* or *Type-2* clones. Mathematically, $T_3^p = T_3 - T_2$, where T_3^p denotes the set of pure *Type-3* clones.

11.2.2 Metrics

The most important metrics used in this study are defined in terms of density of vulnerabilities with respect to (w.r.t.) syntactic blocks of code (BOC) as well as w.r.t. lines of code (LOC). Note that only source lines of code are taken into consideration excluding comments and blank lines.
Density of vulnerabilities w.r.t. BOC in category x **clones**, denoted as ∂_x^β, is defined as the ratio of the number of vulnerabilities found in clones of category x and the number of clones of category x where, category $x \in \{T_1, T_2, T_3, T_2^p, T_3^p\}$. Mathematically,

$$\partial_x^\beta = \frac{\nu_x}{\beta_x} \tag{11.1}$$

Table 11.1: Abridged Description of Major Vulnerabilities Found in the Subject Systems

Vulnerability	**Description**
LawOfDemeter (LD)	Program unit needing too much knowledge about other units.
LocalVariableCouldBeFinal (LVF)	Local variable assigned only once but not declared final.
ShortVariable (SV)	A field, local, or parameter with a too short name.
OnlyOneReturn (OOR)	Method with more than one exit points.
IfStmtsMustUseBraces (ISB)	'if' statements without accompanying curly braces.
AssertionsShouldIncludeMessage (AIM)	Assertions including *no* error message.
UselessParentheses (UP)	Useless parentheses in code.
IfElseStmtsMustUseBraces (IEB)	'if-else' statements without accompanying curly braces.
AvoidInstantiatingObjectsInLoops (AOL)	Instantiation of new objects inside loop.
NullAssignment (NA)	Assignment of a "null" to a variable (outside of its declaration).
ConfusingTernary (CT)	Use of negation within an 'if' expression in 'if-else' statement.
AvoidLiteralsInIfCondition (ALC)	Use of hard coded literals in conditional statements.
MethodShouldUseAnnotation (UA)	Missing annotations for methods.
DataflowAnomaly(DA)	Local definitions and references to variables on different paths.
ModifiedCyclomaticComplexity (MCC)	A variant of Cyclomatic complexity, which treats switch statements as a single decision point.
TooManyMethods (TMM)	A class with too many methods.
NPathComplexity (NPC)	Too high number of acyclic execution paths through a method.
AvoidCatchingGenericException (ACE)	Use of higher level exception in catching low level error conditions.
CommentRequired (CR)	Missing required comment for specific language elements.
MethodArgumentCouldBeFinal (MAF)	Non-final method argument that is never assigned to.
CommentSize (CS)	Dimensions of non-header comments exceeding specified limits.
BeanMembersShouldSerialize (BMS)	Class's member variables not marked as transient, static, or missing accessor methods.
VariableNamingConventions (VNC)	Named of final variables not fully capitalized or use of underscores in names of non-final variables.
LongVariable (LV)	Too long name for a field, method or local variable.
FieldDeclarationsShouldBeAtStartOfClass (FDC)	Class's member fields not declared at the top of the class.
DefaultPackage (DP)	Use of default package private accessibility instead of explicit scoping.
UnusedModifier (UM)	Use modifiers in such a place of code which will be ignored by compiler.
RedundantFieldInitializer (RFI)	Unnecessary explicit initialization of class's member fields.
ImmutableField (IMF)	Class's private fields whose values never change once they are initialized but not made final.

where v_x denotes the number of vulnerabilities found in clones of category x and β_x denotes the total number of clones of category x.

Density of vulnerabilities w.r.t. BOC in all clones, denoted as ∂_c^β, is defined as the ratio of the number of vulnerabilities found in all clones and total the number of all the clones. Mathematically,

$$\partial_c^\beta = \frac{v_c}{\beta_c} \tag{11.2}$$

where v_c denotes the number of vulnerabilities found in all clones and β_c denotes the total number of all categories of clones.

Density of vulnerabilities w.r.t. BOC in non-cloned code, denoted as $\partial_{\bar{c}}^\beta$, is defined as the ratio of the number of vulnerabilities found in non-cloned code and the number of non-cloned blocks of code. Mathematically,

$$\partial_{\bar{c}}^\beta = \frac{v_{\bar{c}}}{\beta_{\bar{c}}} \tag{11.3}$$

where $v_{\bar{c}}$ denotes the number of vulnerabilities found in non-cloned code and $\beta_{\bar{c}}$ denotes the total number of non-cloned blocks of code.

Density of vulnerabilities w.r.t. LOC in category x clones, denoted as ∂_x^ℓ, is defined as the ratio of the number of vulnerabilities found in clones of category x and the number of LOC in all the clones of category x where, category $x \in \{T_1, T_2, T_3, T_2^p, T_3^p\}$. Mathematically,

$$\partial_x^\ell = \frac{v_x}{\ell_x} \tag{11.4}$$

where v_x denotes the number of vulnerabilities found in clones of category x and ℓ_x denotes the total number of LOC in clones of category x.

Density of vulnerabilities w.r.t. LOC in all clones, denoted as ∂_c^ℓ, is defined as the ratio of the number of vulnerabilities found in all clones and the number of LOC in all the clones. Mathematically,

$$\partial_c^\ell = \frac{v_c}{\ell_c} \tag{11.5}$$

where v_c denotes the number of vulnerabilities found in all clones and ℓ_c denotes the total number of LOC in all clones.

Density of vulnerabilities w.r.t. LOC in non-cloned code, denoted as $\partial_{\bar{c}}^\ell$, is defined as the ratio of the number of vulnerabilities found in non-cloned code and the number of LOC in all non-cloned blocks of code. Mathematically,

$$\partial_{\bar{c}}^\ell = \frac{v_{\bar{c}}}{\ell_{\bar{c}}} \tag{11.6}$$

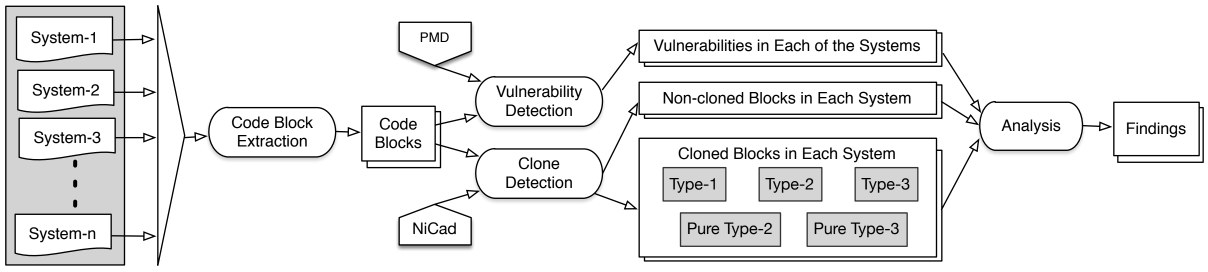


Figure 11.1: Procedural Steps of the Empirical Study

where $v_{\bar{c}}$ denotes the number of vulnerabilities found in non-cloned code and $\ell_{\bar{c}}$ denotes the total number of LOC in non-cloned blocks of code.

Density of a particular vulnerability v in cloned code, denoted as $d_c(v)$, is calculated by dividing the number of instances of v found in cloned code by the total number of LOC in cloned code. Mathematically,

$$d_c(v) = \frac{v_c}{\ell_c} \tag{11.7}$$

where v_c denotes the number of instances of vulnerability v found in cloned code and ℓ_c denotes the total number of LOC across all clones.

Density of a particular vulnerability v in non-cloned code, denoted as $d_{\bar{c}}(v)$, is calculated by dividing the number of instances of v found in non-cloned code by the total number of LOC in non-cloned blocks of code. Mathematically,

$$d_{\bar{c}}(v) = \frac{v_{\bar{c}}}{\ell_{\bar{c}}} \tag{11.8}$$

where $v_{\bar{c}}$ denotes the number of instances of vulnerability v found in non-cloned code and $\ell_{\bar{c}}$ denotes the total number of LOC in all non-cloned blocks of code.

11.3 Study Setup

The procedural steps of our empirical study are summarized in Figure 13.1.

11.3.1 Subject Systems

Our study investigates the source code of 97 software systems of the Qualitas Corpus [281], which is a large curated collection of open source systems of diverse application domains and written in Java.

11.3.2 Clone Detection

Using the `NiCad` [138] clone detector (version 3.5), we separately detect code clones (with at least five LOC) in each of the subject systems. The parameters settings of `NiCad` used in our study are mentioned in Table 12.3. With these settings, `NiCad` detects *Type-1*, *Type-2*, and *Type-3* clones. Further details on `NiCad`'s tuning parameters and their influences on clone detection can be found elsewhere [138]. Then, we compute the pure *Type-2* and pure *Type-3* clones in accordance with their specifications outlined in Section 11.2.

Table 11.2: `NiCad` Settings For Code Clone Detection

Clone Types	`NiCad` Parameter	Value
Type-1	Dissimilarity Threshold	0%
	Identifier Renaming	No Rename
Type-2	Dissimilarity Threshold	0%
	Identifier Renaming	Blind Rename
Type-3	Dissimilarity Threshold	30%
	Identifier Renaming	No Rename

11.3.3 Vulnerability Detection

For the detection of vulnerabilities in source code, we use `PMD` (version 5.3.2) [12], which applies a static rule-based approach for source code analysis and identification of potential vulnerabilities in a software system. For vulnerability detection, we execute `PMD` from command line interface, and feed to it a set of rules, which is the default rule-set packaged with the Eclipse plugin variant of the tool. All others parameters of `PMD` are set to the defaults. Using `PMD`, we separately detect vulnerabilities in each of the subject systems in our study.

11.4 Analysis and Findings

Upon detection of the clones and vulnerabilities, for each of the subject systems, we identify the co-locations of code clones and vulnerabilities, distinguish the vulnerabilities located in non-cloned portion of code, and compute all the metrics described in Section 11.2. To verify the statistical significance of the results derived from our quantitative analysis, we also apply the statistical *Mann-Whitney-Wilcoxon (MWW)* test [234] with $\alpha = 0.05$. The non-parametric *MWW* test does not require normal distribution of data, and thus it suits well for our purpose.

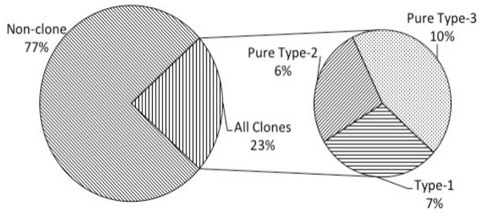


Figure 11.2: Distribution of Vulnerabilities in Cloned and Non-cloned Code

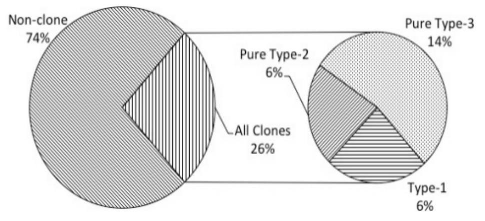


Figure 11.3: Distribution of LOC in Cloned and Non-cloned Code

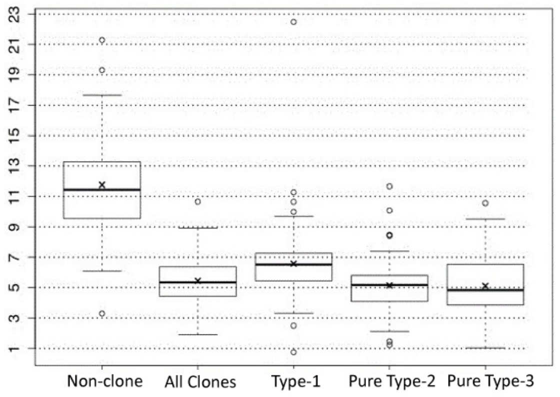


Figure 11.4: Densities of Vulnerabilities w.r.t. BOC

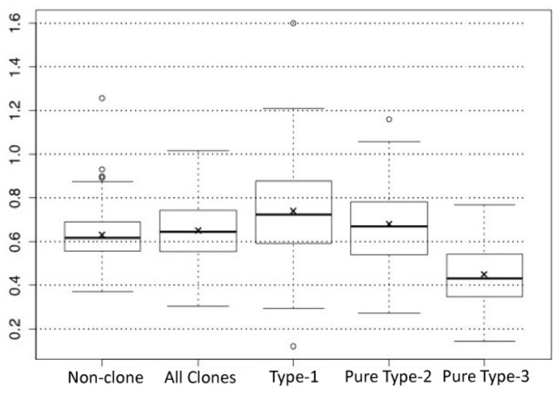


Figure 11.5: Density of Vulnerabilities w.r.t. LOC

11.4.1 Comparative Vulnerability of Cloned vs. Non-Cloned Code

Figure 12.2 presents how the total number of vulnerabilities are distributed in non-cloned code and different types of clones over all the systems. As seen in Figure 12.2, 77% of all vulnerabilities are found in non-cloned source code, whereas the clones contain only 23% of vulnerabilities.

The box-plot in Figure 12.3 presents the densities of vulnerabilities w.r.t. BOC (computed using Equation 12.1, Equation 12.2, and Equation 11.3) found in non-cloned code and in different types of clones over all the subject systems. The 'x' marks in the boxes indicate the mean densities over all the systems. As seen in Figure 12.3, the density of vulnerabilities (w.r.t. BOC) in non-cloned blocks is much higher than that in clones.

Indeed, a larger portion of source code is likely to contain more vulnerabilities than a smaller portion of source code, which might be a reason why non-cloned code seems to have more vulnerabilities as observed in Figure 12.2 and Figure 12.3. To verify this possibility, we compute the distribution of LOC in non-cloned code and different types of clones over all the systems as presented in Figure 12.4. Notice that the distribution of vulnerabilities (Figure 12.2) is very similar to the distribution of LOC (Figure 12.4) in non-cloned code and different types of clones. The average LOC in cloned and non-cloned blocks over all the systems are found to be 8.43 and 19.03 respectively. In the subject systems used in our study, 74% of the source code are clone-free over all the systems as portrayed in Figure 12.4. Thus, the possibility of influence of code size (in terms of LOC) on the number or density of vulnerabilities w.r.t. *BOC* is found to be true.

We, therefore, perform a deeper investigation using the densities of vulnerabilities w.r.t. *LOC*. The box-plot in Figure 12.5 presents the densities of vulnerabilities w.r.t. LOC (computed using Equation 12.3, Equation 12.4, and Equation 11.6) found in non-cloned code and in different types of clones over all the subject systems. Figure 12.5 shows that the densities of vulnerabilities (w.r.t. LOC) in cloned and non-cloned code are almost equal (with a mean difference of 0.02 only). Thus, it appears that there is no significant difference in the density of vulnerabilities w.r.t. LOC in cloned versus non-cloned code. A *MWW* test ($P = 0.28, P > \alpha$) over distribution of densities of vulnerabilities (w.r.t. LOC) across all the subject systems also confirms this finding. Therefore, we derive the answer to the *RQ1* as follows:

Ans. to RQ1: *Cloned code are NOT more vulnerable than non-cloned code. Rather, higher number of vulnerabilities can be found in non-cloned code due to their larger sizes (in terms of LOC) as compared to cloned code.*

11.4.2 Comparative Vulnerability of Different Types of Clones

The distribution of vulnerabilities portrayed in Figure 12.2 shows that the *pure Type-2* clones are found to have the minimum vulnerabilities whereas the number of vulnerabilities found in *Type-1* clones is slightly higher than that in pure *Type-2* clones. The vulnerabilities found in cloned portion of source code are found to be dominated by those found in *pure Type-3* clones. However, the majority of cloned LOC are also in *pure Type-3* clones as can be seen in Figure 12.4, which might be a reason why a higher number of vulnerabilities are found in clones of this particular category.

The box-plot in Figure 12.3 indicates that density of vulnerabilities w.r.t. *BOC* is higher in *Type-1* clones as compared to *pure Type-2* and *pure Type-3*. *MWW* tests between the distributions of vulnerabilities (w.r.t. BOC) in each two of the three categories of clones also suggest statistical significance in the differences except for the case of vulnerabilities in *pure Type-2* and *pure Type-3* clones. The results of the *MWW* tests are presented in Table 11.3.

Table 11.3: *MWW* tests over densities of vulnerabilities w.r.t. *BOC* in different categories of clones

Clone Types	*Type-1*	*Pure Type-2*	*Pure Type-3*
Type-1	-	$P = 0.0$	$P = 0.0$
Pure Type-2	$P = 0.0$	-	$P = 0.7641$
Pure Type-3	$P = 0.0$	$P = 0.7641$	-

In Figure 12.5, the differences in the densities of vulnerabilities w.r.t. *LOC* in the three categories of clones are relatively higher, and the density is the highest in *Type-1* clones while lowest in *pure Type-3*. *MWW* tests between the distributions of vulnerabilities (w.r.t. LOC) in each pair of the three categories of clones also suggest statistical significance in the differences. The results of the *MWW* tests are presented in Table 11.4.

Table 11.4: *MWW* tests over densities of vulnerabilities w.r.t. *LOC* in different categories of clones

Clone Types	*Type-1*	*Pure Type-2*	*Pure Type-3*
Type-1	-	$P = 0.0173$	$P = 0.0$
Pure Type-2	$P = 0.0173$	-	$P = 0.0$
Pure Type-3	$P = 0.0$	$P = 0.0$	-

Based on the findings, we now answer the *RQ2* as follows:

Ans. to RQ2: *Although the clones larger in size (w.r.t LOC) are more vulnerable than smaller clones, in general, Type-1 clones are the most vulnerable while pure Type-3 clones are the least vulnerable and pure Type-2 clones fit in between.*

11.4.3 Relatively Frequent Vulnerabilities

To address the third research question (i.e., *RQ3*), we compute the densities of each individual vulnerability separately in cloned and non-cloned code (over all the subject systems) using Equation 11.7 and Equation 11.8 respectively. Then we distinguish 20 vulnerabilities, which have the highest densities in cloned code over all the subject systems. Let $\mathcal{D}_c$ denote the set of these 20 vulnerabilities. Similarly, we form another set $\mathcal{D}_{\bar{c}}$ consisting of 20 vulnerabilities having the highest densities in non-cloned code. By the union of these two sets we obtain a set $\mathcal{D}$ of 29 vulnerabilities that have the highest densities across both cloned and non-cloned code. Mathematically, $\mathcal{D} = \mathcal{D}_c \cup \mathcal{D}_{\bar{c}}$.

Short descriptions of these 29 vulnerabilities are given in Table 11.1; further elaborations can be found in [12]. The densities of each of these 29 vulnerabilities in cloned and non-cloned code are presented in Table 11.5. Note that, these 29 vulnerabilities represent 85% of total vulnerabilities in cloned code and 89% of the vulnerabilities found in non-cloned code over all the subject systems. Next, we want to partition these vulnerabilities into three clusters: one with vulnerabilities dominating in cloned code, another with those dominating in non-cloned code and a third cluster with vulnerabilities, which almost equally appear in both cloned and non-cloned code.

Cluster Analysis: For the purpose of aforementioned partitioning, we conduct a clustering analysis on the densities of these vulnerabilities in cloned and non-cloned code. For each v of these 29 vulnerabilities, we compute a ratio $\mathcal{M}(v)$ as follows:

$$\mathcal{M}(v) = \frac{d_c(v)}{d_{\bar{c}}(v)}, \ where, \ v \in \mathcal{D} \tag{11.9}$$

The ratios computed for each of the 29 vulnerabilities are presented in the second column from the right in Table 11.5. Notice that, for a particular vulnerability v, the ratio $\mathcal{M}(v)$ close to 1.0 indicates that the vulnerability v almost equally appears in both cloned and non-cloned code. If $\mathcal{M}(v)$ is much higher than 1.0, the appearance of vulnerability v can be characterized to have dominated in cloned code. Similarly, $\mathcal{M}(v)$ being much lower than 1.0 implies that the vulnerability v appears more in non-cloned code. However, a threshold scheme seems required to determine when the value of $\mathcal{M}(v)$ can be considered significantly close to or distant from 1.0.

Instead of setting an arbitrary threshold by ourselves, we apply unsupervised *Hierarchical Agglomerative Clustering* [282] for partitioning the values of $\mathcal{M}(v)$. The dendrogram produced from this statistical clustering is presented in Figure 11.6. In the dendrogram, three major clusters are evident, two marked with dotted rectangles and the third left unmarked in the middle. The values of $\mathcal{M}(v)$ for the vulnerabilities in the middle cluster range between 0.97 and 1.92. This middle cluster

Table 11.5: Vulnerabilities Dominating in Cloned and Non-cloned Code

Vulner- -ability (v)	**Density** **Clone** $d_c(v)$	**Non-clone** $d_{\bar{c}}(v)$	**Ratio** $\mathcal{M}(v) = \frac{d_c(v)}{d_{\bar{c}}(v)}$	**High Frequency Area**
LD	0.1489	0.10547	1.4118	**Both** clone and non-clone code
LVF	0.08561	0.05885	1.4547	
SV	0.02258	0.02149	1.0507	
OOR	0.03129	0.01625	1.9255	
ISB	0.02124	0.01605	1.3234	
AIM	0.00968	0.00707	1.3692	
UP	0.00822	0.00602	1.3654	
IEB	0.00472	0.00485	0.9732	
AOL	0.00428	0.00301	1.4219	
NA	0.00423	0.00258	1.6395	
CT	0.00417	0.00239	1.7448	
ALC	0.00406	0.00274	1.4818	
UA	0.00389	0.00218	1.7844	
DA	0.0563	0.02362	2.3836	**Clone** code only
MCC	0.01026	0.0013	7.8923	
TMM	0.00421	0.0002	21.0500	
NPC	0.00398	0.00087	4.5747	
ACE	0.00389	0.00127	3.0630	
CR	0.03638	0.08105	0.4489	**Non-clone** code only
MAF	0.04668	0.08102	0.5762	
CS	0.00227	0.04542	0.0500	
BMS	0.00006	0.02532	0.0024	
VNC	0.00277	0.01866	0.1484	
LV	0.00309	0.01727	0.1789	
FDC	0.00002	0.00872	0.0023	
DP	0.0033	0.00831	0.3971	
UM	0.00002	0.00517	0.0039	
RFI	0.00004	0.00467	0.0086	
IMF	0.00001	0.00444	0.0023	

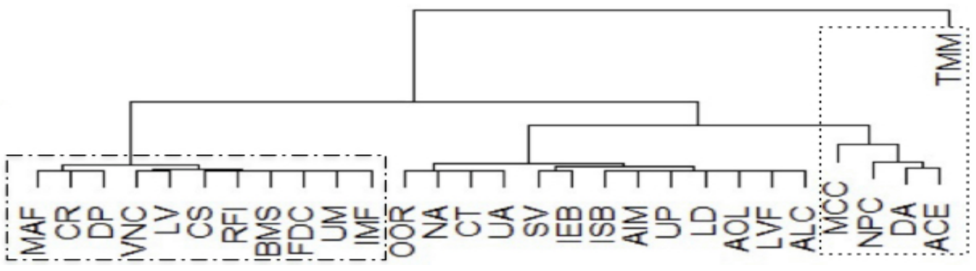


Figure 11.6: Hierarchical Agglomerative Clustering of Vulnerabilities

includes a set of those vulnerabilities, which equally appear in *both* cloned and non-cloned codes. Let G_b denote this cluster.

For all of the five vulnerabilities (i.e., MCC, NPC, DA, ACE, and TMM) in the right-most cluster $\mathcal{M}(v) \geq 2.38$, which indicates that these vulnerabilities appear more frequently in *cloned* code compared to their presence in non-cloned clone. Let G_c denote the cluster of these vulnerabilities. The left most cluster, denoted as, $G_{\bar{c}}$, includes the vulnerabilities with $\mathcal{M}(v) < 0.97$, and they are frequently found in *non-cloned* code. The right-most column in Table 11.5 labels the vulnerabilities in accordance with how they are clustered here.

Statistical Significance: For each of the three clusters of vulnerabilities, we separately conduct *MWW* tests between the densities of those vulnerabilities in cloned and non-cloned code to determine the statistical significance of the difference in their existence in those two categories (i.e., cloned and non-cloned) of code. The results of the separate *MWW* tests over each of the clusters are presented in Table 11.6.

Table 11.6: *MWW* tests between density-distributions in cloned and non-cloned code for vulnerabilities of each cluster

Cluster	G_c	$G_{\bar{c}}$	G_b
***P*-values**	0.0474	0.0024	0.3575

The *P*-values in Table 11.6 indicate statistical significance in the differences of density-distribution in cloned and non-cloned code for vulnerabilities in cluster G_c and $G_{\bar{c}}$, but not for those in cluster G_b. Thus, our clustering of the vulnerabilities is confirmed accurate with statistical significance. Now, we answer the research question *RQ3* as follows:

Ans. to RQ3: *There are distinct sets of vulnerabilities (as characterized in Table 11.5), which frequently appear in cloned code or non-cloned code, while many other vulnerabilities are found to be equally present in both cloned and non-cloned code.*

11.5 Threats to Validity

In this section, we discuss possible threats to the validity of our study and how we have mitigated their effects.

Construct Validity: In the detection of vulnerabilities with `PMD`, we used its default settings and relied on the set of rules, which came with the Eclipse plug-in variant of the tool. Those set of rules might not have covered all possible vulnerabilities, and we considered each vulnerability equally important. In the selection of the two dominant sets of vulnerabilities (Section 11.4.3), we picked top 20 vulnerabilities for each set having the highest densities in cloned and non-cloned code. Although those chosen vulnerabilities cover more than 80% of all distinct vulnerabilities and more than 85%

instances of vulnerabilities found in all the systems, this choice may still be considered as a threat to validity of this work.

Internal Validity: The clone detector, `NiCad`, used in our study, is reported to be very accurate in clone detection [138], and we have carefully set `NiCad`'s parameters. The tool, `PMD`, used in our work for vulnerability detection, is also known effective and widely used in both industry and research community. However, 15 out of 112 systems in the Qualitus Corpus [281] were failed to be processed by either `NiCad` or `PMD`. Those 15 systems are excluded from our study. Moreover, we manually verified the correctness of computations for all the metrics used in our work. Thus, we develop high confidence in the internal validity of this study.

External Validity: Although our study includes a large number of subject systems, all the systems are open-source and written in Java. Thus the findings from this work may not be generalizable for industrial systems and source code written in languages other than Java.

Reliability: The methodology of this study including the procedure for data collection and analysis is documented in this chapter. The subject systems being open-source, are freely accessible while the tools `NiCad` and `PMD` are also available online. Therefore, it should be possible to replicate the study.

11.6 Related Work

It is often believed that inconsistent changes to clones cause program faults and frequent changes may lead to significant instances of inconsistent changes [83, 72]. Thus, to develop an understanding on the fault-proneness of code clones, studies have been conducted to examine the stability (in terms of frequency and sizes of changes) and inconsistent changes in evolving code clones.

Juergens et al. [72] reported that inconsistent changes to clones are very frequent and a significant number of faults are induced by such inconsistent changes. Barbour et al. [83] suggested that late propagations due to inconsistent changes are prone to introduce software defects. While Lozano and Wermelinger [73] suggested that having a clone may increase the maintenance effort for changing a method, Hotta et al. [79] reported code clones not to have any negative impact on software changeability. Lozano et al. [91] reported that a vast majority of methods experience larger and frequent changes when they contain cloned code. Mondal et al. [82] also reported code clones to be less stable. However, opposite results are found from the other studies [283, 90, 78, 80].

Attempts are also made to explore fault-proneness of clones by relating them with bug-fixing changes obtained from commit history. Such a study was conducted by Jingyue et al. [87], who reported that only 4% of the bugs were found in duplicated code. In a similar study, Rahman et al. [25] also observed that majority of bugs were not significantly associated with clones. These findings contradict with those of Juergens et al. [72] and Barbour et al. [83].

The contradictory results from the earlier studies imply the necessity of further comparative investigations from a different dimension, which is exactly what we have done in this study. We have carried out a comparative investigation of vulnerabilities in clones and non-coned code, which was missing in the literature. In addition, the comparative analysis of vulnerabilities in *Type-1*, *pure Type-2*, and *pure Type-3* clones is another unique aspect of our work.

11.7 Summary

In this chapter, we have presented a quantitative empirical study on the vulnerabilities (in terms of bad coding patterns) in different types of code clones and non-cloned code in 97 open-source software systems written in Java. To the best of our knowledge, no other work in the past conducted a comparative study of such vulnerabilities in cloned and non-cloned as done in our work. In our study, we have found no significant differences between the densities of vulnerabilities in code clones and clone-free source code. Surprisingly, among the three categories (i.e., *Type-1*, *pure Type-2*, and *pure Type-3*) of clones studied in our work, *Type-1* clones are found to be the most vulnerable whereas *pure Type-3* are the least. In addition, our study identifies a set of five vulnerabilities that appear more frequently in cloned code compared to non-cloned code. Another set of 11 vulnerabilities are also distinguished, which are more frequently found in non-cloned code as opposed to cloned code. The results are validated in the light of statistical significance.

The findings from this study significantly advance our understanding of the characteristics, impacts, and implications of code clones in software systems. These findings can help in identifying problematic clones, which demand extra care and those vulnerabilities about which the developers need to be particularly cautious about while reusing code by cloning. For example, since *Type-1* clones are found to be the most vulnerable, and that refactoring of *Type-1* clones can be expected to be easier (due to absence of much differences among them) than refactoring other types of clones, we argue that this particular type of clones should be removed from source code by frequent refactoring.

Chapter 12

Security Vulnerabilities in Clones and Non-Cloned Code

In the last chapter (Chapter 11), we presented a comparative study on different types of clones (i.e., copy-pasted code) and non-cloned code on the basis of their code-smells. Although earlier studies examined the bug-proneness, stability, and changeability of clones against non-cloned code, *the security aspects remained ignored*. In this chapter, we present a study to explore and understand the security vulnerabilities and their severity in different types of clones compared to non-clone code.

The rest of the chapter is organized as follows. In Section 12.1, we introduce the motivation and context of the work. Section 12.2 introduces the terms and metrics used in this study. Section 12.3 describes the procedure of our study. Analyses and findings are presented in Section 12.4. Section 12.5 presents the possible threats to the validity of this work. In Section 12.6, we discuss the existing work relevant to ours. Finally, Section 12.7 concludes the chapter with directions for future work.

12.1 Introduction

Software security has become one of the most pressing concerns recently. Software developers are expected to write secure source code and minimize security vulnerabilities in the systems under development. However, the developers' copy-paste practice for code reuse often cause the multiplication and propagation of program faults and security vulnerabilities [95, 72, 75]. Thus, there is a possibility that such vulnerabilities can exist in cloned code at a higher rate.

Code clone (i.e., similar or duplicated code) is already identified as a notorious code smell (i.e., a symptom indicating source of future problems) [74] that cause other problems such as reduced code quality, code inflation, and change difficulties [72, 73, 284, 75]. Nevertheless, software systems typically have 9%-17% [68] cloned code, and the proportion is sometimes found to be even 50% [69] or higher [70].

Clones are arguably a major contributor to the high maintenance cost for software systems, and as much as 80% of software costs are spent on maintenance [76]. Thus, to understand the characteristics and contexts of the detrimental impacts of clones, earlier studies examined the comparative stability of clones as opposed to non-cloned code [78, 79, 80, 81, 82], relationships of clones with bug-fixing changes [83, 84, 85, 72, 86, 87, 25, 88, 89], the impacts of clones on program's changeability [90, 73, 91] and comparative fault-proneness of cloned and non-cloned code [95, 285].

However, the security aspects of code clones have never been studied before, although the reuse of vulnerable components and source code (i.e., code clones) multiply security vulnerabilities [92, 93, 94]. This chapter presents, a large empirical study on the security vulnerabilities in code clones and presents a comparative analysis of these vulnerabilities in different types of clones as opposed to non-cloned code. In particular, we address the following five research questions.

RQ1: *Do code clones contain higher number of security vulnerabilities than non-cloned code or vice versa*?

— The answer to this question will add to our understanding of the negative impacts of clones, which will be useful in *cost-benefit analysis* [286] for improved clone management.

RQ2: *Do clones of a certain category contain more security vulnerabilities than others?*

— If a certain type of clones are found to have higher number of security vulnerabilities, those clones will be high-priority candidates for removal or careful maintenance.

RQ3: *Do code clones contain more severe (i.e. riskier) vulnerabilities compared to non-cloned code or vice versa*?

— All the security vulnerabilities are not equally risky in terms of security threat. The result of this research question will advance our understanding of clones' impacts and will be useful in cost-effective clone management [286].

RQ4: *Do clones of a certain category contain relatively severe (i.e., riskier) security vulnerabilities than others?*

— If a certain type of clones are found to have riskier security vulnerabilities, those clones will demand especial attention and high-priority in clone-removal process.

RQ5: *Can we distinguish some security vulnerabilities that appear more frequently in cloned code as opposed to non-cloned code?*

— If we can distinguish a set of vulnerabilities that appear more frequently in code clones, the finding will help software developers to stay cautious of such vulnerabilities while cloning source code. In addition, that particular set of vulnerabilities can be minimized by clone refactoring.

To answer the aforementioned research questions, we conduct a large-scale empirical study over 8.7 million lines of source code in 34 open-source software systems written in C. Using a wide range of metrics and characterization criteria, we carry out in-depth quantitative analyses on the source code of the systems with respect to different categories of code clones, non-cloned code, and a set of security vulnerabilities. In this regard, this work makes the following two contributions:

- We present a large-scale comparative study of the security vulnerabilities in code clones and non-cloned code. To the best of our knowledge, no such study of the security vulnerabilities in code clones exists in the literature.

- We also perform a comparative analysis of security vulnerabilities in different types of clones (e.g., *Type-1*, *Type-2*, and *Type-3*), which informs the relative security implications of clones at different similarity levels.

12.2 Terminology and Metrics

In this section, we describe and define the terminologies and metrics that are used in our work. Some of the metrics are adapted from the literature [95, 67].

12.2.1 Security Vulnerabilities

A software security vulnerability is defined as a weakness in a software system that can lead to a compromise in integrity, availability or confidentiality of that software system. For example, *buffer overflow* and *dangling pointers* are two well known security vulnerabilities. The cyber security community maintains a community-developed list of common software security vulnerabilities where each category of vulnerability is enumerated with a CWE (Common Weakness Enumeration) number [108]. For example, CWE-120 refers to those vulnerabilities that fall into the CWE category of *classic buffer overflow*. More examples of security vulnerabilities along with their CWE enumerations are presented in Table 12.1.

12.2.2 Metrics

The required metrics are defined in terms of density of vulnerabilities with respect to (w.r.t.) syntactic blocks of code (BOC) as well as w.r.t. lines of code (LOC). Only source lines of code are taken into consideration excluding comments and blank lines.

Let, C denote the set of all *cloned* code blocks and $\bar{C}$ denote the set of all *non-cloned* code blocks.

Also let, $\mathcal{V}_C$ denote the set of vulnerabilities found in C and $\mathcal{V}_{\bar{C}}$ denote the set of vulnerabilities located in $\bar{C}$. A cloned code block in C can be of category $\mathcal{X}$ clone where $\mathcal{X} \in \{T_1, T_2, T_3, T_2^p, T_3^p\}$. Thus, $\mathcal{V}_C$ can be split into multiple sets with $\mathcal{V}_{\mathcal{X}}$ denoting the set of vulnerabilities identified in clones of category $\mathcal{X}$.

Density of vulnerabilities w.r.t. BOC in category $\mathcal{X}$ clones, denoted as $\partial_{\mathcal{X}}^{\beta}$, is defined as the ratio of the number of vulnerabilities found in clones of category $\mathcal{X}$ to the number of cloned blocks in category $\mathcal{X}$ clones. Mathematically,

$$\partial_{\mathcal{X}}^{\beta} = \frac{|\mathcal{V}_{\mathcal{X}}|}{\beta_{\mathcal{X}}} \tag{12.1}$$

where $|\mathcal{V}_{\mathcal{X}}|$ denotes the number of vulnerabilities found in the blocks of category $\mathcal{X}$ clones and $\beta_{\mathcal{X}}$ denotes the total number of block clones of category $\mathcal{X}$. And, $\mathcal{X} \in \{T_1, T_2, T_3, T_2^p, T_3^p\}$.

Table 12.1: Security vulnerabilities frequently identified in the systems

Security Vulnerability	**Description**
Buffer Overflow	An application attempts to write data past the end of a buffer (CWE-120).
Uncontrolled Format String	Submitted data of an input string is evaluated as a command by the application (CWE-134).
Integer Overflow	The result of an arithmetic operation exceeds the maximum size of the integer type used to store it (CWE-190).
Null Pointer Dereference	Dereference a pointer that is null (CWE-476).
Memory Leak	Not release allocated memory (CWE-401).
Null Termination Errors	A string is incorrectly terminated (CWE-170).
Concurrency Errors	Concurrent execution using shared resource with improper synchronization (CWE-362).
Double Free	The program calls free() twice on the same memory address (CWE-415).
Access Freed Memory	Access memory after it has been freed (CWE-416).
Insecure Randomness	The software may use insufficiently random numbers or values in a security context that depends on unpredictable numbers (CWE-330).
OS Command Injection	Improper neutralization of special elements used in an operating systems command (CWE-78).
Insecure Temporary File	Creating and using insecure temporary files can leave application and system data vulnerable to attack (CWE-377).
Reliance on Untrusted Inputs	The input can be modified by an untrusted actor in a way that bypasses the protection mechanism (CWE-807).
Poor Code Quality	Indication that the product has not been carefully developed or maintained (CWE-398).
Dead Code	Code that can never be executed (CWE-561).
Use of Obsolete Functions	The code uses deprecated or obsolete functions, which suggests that the code has not been actively reviewed or maintained (CWE-477).
Risky Cryptography	Stealing the information protected, under normal conditions, by the SSL/TLS encryption (CWE-327).
Unused Variable	The variable's value is assigned but never used (CWE-563).

Density of vulnerabilities w.r.t. BOC in type $\mathcal{T}$ code, denoted as $\partial^{\beta}_{\mathcal{T}}$, is defined as the ratio of the number of vulnerabilities found in type $\mathcal{T}$ code to total number of blocks in type $\mathcal{T}$ code, where $\mathcal{T} \in \{\mathcal{C}, \bar{\mathcal{C}}\}$. Mathematically,

$$\partial^{\beta}_{\mathcal{T}} = \frac{|\mathcal{V}_{\mathcal{T}}|}{\beta_{\mathcal{T}}} \tag{12.2}$$

where $\mathcal{V}_{\mathcal{T}} \in \{\mathcal{V}_{\mathcal{C}}, \mathcal{V}_{\bar{\mathcal{C}}}\}$ and $\beta_{\mathcal{T}}$ denotes the total number of blocks in type $\mathcal{T}$ code.

Density of vulnerabilities per 1,000 LOC (KLOC) in category $\mathcal{X}$ clones, denoted as $\partial^{\ell}_{\mathcal{X}}$, is defined as follows:

$$\partial^{\ell}_{\mathcal{X}} = \frac{|\mathcal{V}_{\mathcal{X}}|}{\ell_{\mathcal{X}}} * 1000 \tag{12.3}$$

where $\ell_{\mathcal{X}}$ denotes the total number of LOC in clones of category $\mathcal{X}$.

Density of vulnerabilities per KLOC in type $\mathcal{T}$ code, denoted as $\partial^{\ell}_{\mathcal{T}}$, is defined as follows:

$$\partial^{\ell}_{\mathcal{T}} = \frac{|\mathcal{V}_{\mathcal{T}}|}{\ell_{\mathcal{T}}} * 1000 \tag{12.4}$$

where $\ell_{\mathcal{T}}$ denotes the total number of LOC in type $\mathcal{T}$ code.

Risk severity score per KLOC in category $\mathcal{X}$ clones, denoted as $\mathcal{R}_{\mathcal{X}}$, is defined as the ratio of sum of severity scores of the vulnerabilities found in clones of category $\mathcal{X}$ to the number of KLOC in the clones of category $\mathcal{X}$. Mathematically,

$$\mathcal{R}_{\mathcal{X}} = \frac{\sum_{v \in \mathcal{V}_{\mathcal{X}}} s(v)}{\ell_{\mathcal{X}}} * 1000 \tag{12.5}$$

where $s(v)$ denotes the severity score of vulnerability v.

Risk severity score per KLOC in type $\mathcal{T}$ code, denoted as $\mathcal{R}_{\mathcal{T}}$, is defined as the ratio of sum of severity scores of the vulnerabilities found in type $\mathcal{T}$ code to the number of KLOC in that type of code. Mathematically,

$$\mathcal{R}_{\mathcal{T}} = \frac{\sum_{v \in \mathcal{V}_{\mathcal{T}}} s(v)}{\ell_{\mathcal{T}}} * 1000 \tag{12.6}$$

Density of a particular group of vulnerabilities $\mathcal{G}$ per KLOC in type $\mathcal{T}$ code, denoted as $\partial^{\mathcal{G}}_{\mathcal{T}}$, is calculated by dividing the number of vulnerabilities found in CWE category $\mathcal{G}$ in type $\mathcal{T}$ code by the total number of KLOC in that type of code. Mathematically,

$$\partial^{\mathcal{G}}_{\mathcal{T}} = \frac{|\mathcal{G}_{\mathcal{T}}|}{\ell_{\mathcal{T}}} * 1000 \tag{12.7}$$

where $|\mathcal{G}_{\mathcal{T}}|$ denotes the number of vulnerabilities belong to a CWE category $\mathcal{G}$ found in type $\mathcal{T}$ code.

12.3 Study Setup

The procedural steps of our empirical study are summarized in Figure 13.1.

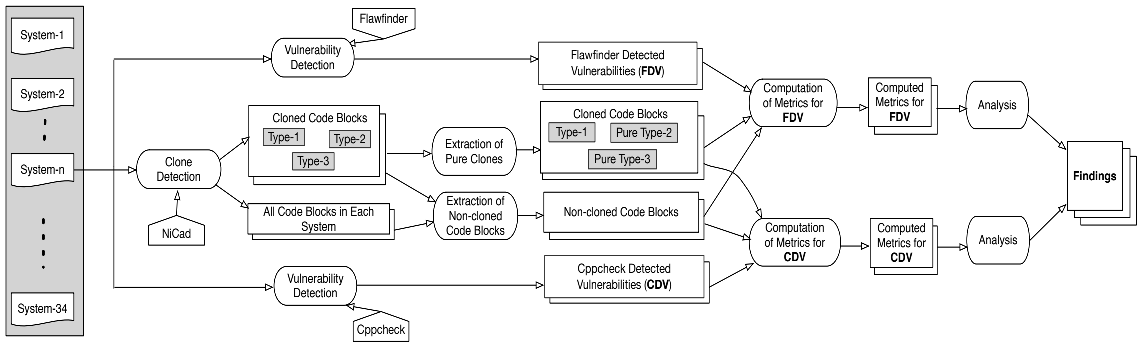


Figure 12.1: Procedural Steps of the Empirical Study

12.3.1 Subject Systems

Our study investigates the source code of 34 open-source software systems written in C. Although some of the systems contain files written in other languages such as C++, Perl and other scripting languages, we only consider those files, which have extension '.c' or '.h' to exclude source code written in languages other than C. Projects of various sizes are deliberately chosen from different application domains including networking, communication, security, and text editing. Most of these subject systems are well-reputed in their respective development ecosystems (e.g., GitHub and SourceForge) and used in earlier research studies [287, 288, 289, 290, 291].

The names and sizes of the subject systems in LOC in clones and non-clone code are presented in Table 12.2. In computation of sizes, only the source code written in C are considered. The average size of the subject systems is 256 thousand LOC where the largest project consists of 3.4 million LOC and the smallest one contains 16 thousand LOC.

12.3.2 Code Clone Detection

Using the `NiCad` [138] clone detector (version 3.5), we separately detect code clones in each of the subject systems at the granularity of syntactic code blocks. The parameters settings of `NiCad` used in our study are mentioned in Table 12.3. With these settings, `NiCad` detects *Type-1*, *Type-2*, and *Type-3* clones. Further details on `NiCad`'s tuning parameters and their influences on clone detection can be found elsewhere [138]. Then, we compute the *pure Type-2* and *pure Type-3* clones in accordance with their definitions outlined in Section 12.2.

Table 12.2: Subject systems and their LOC in cloned and non-cloned code

Subject System	**# of LOC written in C only**		
	Non-clone	**Clone**	**Total**
Asn1c	40,487	5,488	45,975
Atlas	369,944	116,586	486,530
Clamav	337,019	40,371	377,390
Claws	239,784	32,868	272,652
Conky	27,759	14,494	42,253
Courier	107,223	13,105	120,328
Emacs	314,521	17,092	331,613
Ettercap	36,268	5,182	41,450
Ffdshow	832,342	46,367	878,709
Freediag	15,225	1,380	16,605
Freedroid	60,828	3,957	64,785
Gedit	42,112	4,331	46,443
Glimmer	29,536	4,923	34,459
Gnuplot	86,851	6,712	93,563
Gretl	302,777	36,499	339,276
Grisbi	99,205	18,479	117,684
Ipsec-tools	60,728	10,222	70,950
Modsecurity	26,332	7,232	33,564
Nedit	85,001	9,506	94,507
Net-snmp	231,422	42,725	274,147
Ocf-linux	45,761	8,743	54,504
Opendkim	47,298	18,480	65,778
Opensc	119,646	11,823	131,469
Putty	78,918	11,322	90,240
Razorback	36,403	8,823	45,226
Sdcc	3,477,010	4,122	3,481,132
Tboot	21,942	6,115	28,057
Tcl8	326,217	37,693	363,910
Tcpreplay	43,191	4,740	47,931
Trousers	56,673	17,599	74,272
Vi	21,924	734	22,658
Vim	314,480	5,752	320,232
XSupplicant	78,441	24,028	102,469
Zabbix	107,475	18,156	125,631

Table 12.3: `NiCad` Settings For Code Clone Detection

Chosen Parameters for `NiCad`	**Target Clone Types**		
	Type-1	**Type-2**	**Type-3**
Dissimilarity Threshold	0.0	0.0	0.3
Identifier Renaming	no rename	blind rename	no rename
Granularity	block	block	block
Minimum Clone Size	5 LOC	5 LOC	5 LOC

12.3.3 Security Vulnerability Detection

For the detection of vulnerabilities in source code, we use two open-source static analysis tools `Flawfinder` (version 1.3) [9] and `Cppcheck` (version 1.76.1) [13]. Their ability to detect different sets of vulnerabilities make them appropriate for our analysis. For example, `Flawfinder` is capable of detecting vulnerabilities such as *Uncontrolled Format String*, *Integer Overflow* and *Use of Risky Cryptographic Algorithm*, which `Cppcheck` fails to detect [292]. On the other hand, `Cppcheck` is able to detect vulnerabilities such as *Memory Leak*, *Dead Code* and *Null Pointer Dereference* that `Flawfinder` cannot detect [292].

We now briefly describe how these tools work for the detection of security vulnerabilities and the ways these tools are used in our study. We also present justifications for choosing these tools for our study.

12.3.3.1 `Flawfinder`

The tool contains a database of common functions known to be vulnerable. It operates by performing lexical tokenization of the C/C++ code and comparing the tokens with those in the database. Once the comparison is performed, it reports a list of possible vulnerabilities along with a source code line number and a numeric risk-level (i.e., severity score) associated each of the detected vulnerabilities. The severity scores vary from one (indicating little security risk) to five (indicating high risk).

Although many other open-source static analysis tools such as, `RATS` [293], `SPLINT` [14] and `ITS4` [294] exist to identify potential security issues, we choose `Flawfinder` for a number of reasons. This tool is reported to have the highest vulnerabilities detection rate (i.e., highest recall) among all the existing security vulnerability detectors [292, 295, 296, 297]. Moreover, a comparison of vulnerability detection tools [295] also recommend choosing `Flawfinder` to detect security vulnerabilities. `Flawfinder` is also widely used in many earlier studies [297, 298, 296].

To detect vulnerabilities with `Flawfinder`, we execute the tool from command line interface and separately detect vulnerabilities in each of the subject systems in our study. We refer to the vulnerabilities detected using `Flawfinder` as $\mathcal{FDV}$ (`Flawfinder` Detected Vulnerabilities).

12.3.3.1.1 Limiting false positives in `Flawfinder` Although `Flawfinder` has the highest detection rate, at times, it is blamed for reporting many false positives [296]. To reduce the false positives, we alter the default configuration of `Flawfinder`. We run the tool with *'-F'* configuration parameter that reduces 62% of the false positives as reported in a controlled experiment [299]. To further reduce the effect of false positives, we discard any detected vulnerabilities associated with risk-levels less than two.

Table 12.4: Reduction of false positives in vulnerability detection using the customized configuration of `Flawfinder`

Test Suite ID	# of False Positives Reported		Reduction of False Positives
	Default Config.	Customized Config.	
57	08	01	87.50%
58	10	04	60.00%

12.3.3.1.2 Effectiveness of customized configuration To determine the effectiveness of the customized configuration stated above, we collect two C/C++ test suites, *test-suite-57* and *test-suite-58* from *Software Assurance Reference Dataset* (SARD) [300]. These test suites include 41 and 39 pieces of code respectively. While all pieces of code in *test-suite-57* are known to be vulnerable, none of the pieces of code in *test-suite-58* are vulnerable.

We run `Flawfinder` on the two test suites separately using both default and customized configurations. By comparing the reported vulnerabilities with the known vulnerabilities in the test suites, we compute false positives w.r.t. both test suites for each of the configurations and present the result in Table 12.4. Notice that with the customized configuration, false positives are reduced by 87.5% and 60% for *test-suite-57* and *test-suite-58* respectively. We also observe 30% reduction in the detection of vulnerabilities mostly due to the elimination of false positives using the customized configuration for *test-suite-57*. Thus, at the cost of a minor sacrifice in recall, the customized configuration is able to reduce significant number of false positives.

12.3.3.2 `Cppcheck`

`Cppcheck` supports a wide variety of static checks that are rigorous, rather than heuristic in nature [13]. `Cppcheck` is developed aims to report zero false positives [13], which makes it unique from other static security analysis tools. Unlike `Flawfinder`, `Cppcheck` does not assign numeric risk-levels to vulnerabilities. Instead, `Cppcheck` classifies vulnerabilities into six severity categories namely, *Error, Warning, Style, Performance, Portability,* and *Information*. We operate `Cppcheck` from command line using its default configuration separately on each of the subject systems. The output of the tool is generated in XML. We refer to the security vulnerabilities detected by `Cppcheck` as $\mathcal{CDV}$ (`Cppcheck` Detected Vulnerabilities).

Brief description along with CWE numbers of the major vulnerabilities detected in the subject systems using `Flawfinder` and `Cppcheck` are presented in Table 12.1. Those reported as vulnerabilities but do not have an associated CWE number are excluded from our analysis to further limit effects of possible false positives.

Table 12.5: Detected security vulnerabilities and their severity in cloned and non-cloned code

Subject System	**# of $\mathcal{FDV}$**		**# of $\mathcal{CDV}$**		**Severity score**	
	$\mathcal{NC}$	$\mathcal{C}$	$\mathcal{NC}$	$\mathcal{C}$	$\mathcal{NC}$	$\mathcal{C}$
Asn1c	119	10	183	18	309	70
Atlas	1,258	1,607	2,307	3,240	4,429	4,246
Clamav	1,151	166	6,379	1,901	2,689	345
Claws	429	35	1,496	138	1,148	108
Conky	310	69	10	41	850	200
Courier	2,569	270	1,152	67	7,479	900
Emacs	1,071	99	666	108	2,990	281
Ettercap	419	78	235	23	982	177
Ffdshow	1,306	158	10,042	424	1,457	330
Freediag	416	54	104	11	1,521	202
Freedroid	429	26	180	0	1,181	88
Gedit	21	9	276	7	51	18
Glimmer	151	19	326	214	450	64
Gnuplot	717	67	1,322	87	2,172	208
Gretl	2,850	359	2,992	249	8,847	1,116
Grisbi	54	40	293	27	177	82
Ipsec-tools	447	100	790	134	960	210
Modsecurity	163	22	193	40	308	44
Nedit	592	62	624	56	1,884	216
Net-snmp	1,715	352	2,002	177	4,071	763
Ocf-linux	153	17	379	61	245	134
Opendkim	201	40	296	98	383	146
Opensc	1,150	141	735	30	2,347	404
Putty	523	108	593	59	1,217	392
Razorback	104	31	359	39	261	94
Sdcc	26	9	291	38	182	32
Tboot	105	59	80	225	234	140
Tcl8	1,111	218	3,592	463	2,552	628
Tcpreplay	421	87	232	10	1,252	228
Trousers	263	110	460	34	560	224
Vi	144	4	403	16	417	52
Vim	805	22	2,336	43	2,264	70
XSupplicant	838	246	2,593	505	1,822	548
Zabbix	512	82	566	24	992	278

Here, $\mathcal{NC}$ = non-cloned code and $\mathcal{C}$ = cloned code

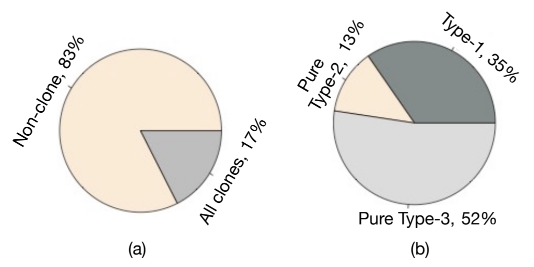


Figure 12.2: Distribution of $\mathcal{FDV}$ (a) in Cloned and Non-cloned Code and (b) in Different Types of Clones

12.4 Analysis and Findings

After detecting clones and vulnerabilities in the software systems, we determine locations of the detected vulnerabilities in different types of code (i.e., cloned and non-cloned code). A vulnerability is said to be located in cloned code if the reported source code line number of that vulnerability included in a cloned block, otherwise, the vulnerability is located in non-cloned block. For each of the subject systems, we identify the co-locations of code clones and vulnerabilities, distinguish the vulnerabilities located in non-cloned portion of code, and compute all the metrics described in Section 12.2. The number of $\mathcal{FDV}$, $\mathcal{CDV}$ in the clones and non-cloned code in each of the subject systems, and the cumulative vulnerability severity scores (obtained from `Flawfinder`) are presented in Table 12.5.

We separately analyze the security vulnerabilities detected using both `Flawfinder` and `Cppcheck` to derive answers to the research questions RQ1, RQ2, and RQ5. Since `Cppcheck` does not provide the numeric severity score to indicate risk-level of a security vulnerability, the research questions RQ3 and RQ4 are addressed using $\mathcal{FDV}$ only.

Statisitical measurements. To verify the statistical significance of the results derived from our analyses, we apply the statistical *Mann-Whitney-Wilcoxon (MWW)* test [234] and *Kruskal-Wallis* test [234] at the significance level $\alpha = 0.05$. We perform *Kruskalmc* [234] test for post-hoc analysis at the same significance level. As the non-parametric *MWW*, *Kruskal-Wallis* and *Kruskalmc* tests do not require normal distribution of data, those tests suit well for our purpose. To measure the effect size, we compute the non-parametric effect size *Cliff's delta* [234].

12.4.1 Vulnerabilities in Clones vs. Non-Cloned Code

Analysis Using $\mathcal{FDV}$: Figures 12.2(a) and 12.2(b) present the distributions of vulnerabilities detected using `Flawfinder` in non-cloned code and in different types of clones respectively over all the systems. As seen in Figure 12.2(a), 83% of all the vulnerabilities are found in non-cloned source code, whereas the clones contain only 17% of vulnerabilities.

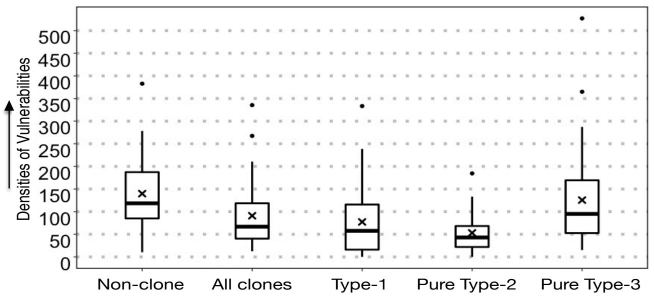


Figure 12.3: Densities of $\mathcal{FDV}$ w.r.t. BOC

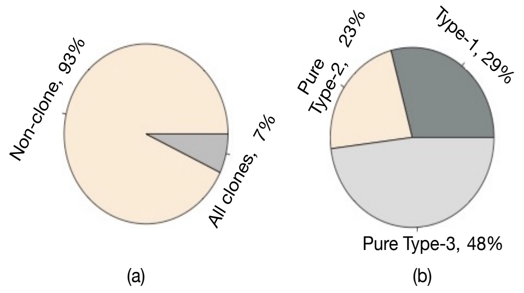


Figure 12.4: Distribution of LOC (a) in Cloned and Non-cloned Code and (b) in Different Types of Clones

The box-plot in Figure 12.3 presents the densities of vulnerabilities (i.e. $\mathcal{FDV}$) w.r.t. BOC (computed using Equation 12.1 and Equation 12.2) found in non-cloned code and in different types of clones over all the subject systems. The 'x' marks in the boxes indicate the mean densities over all the systems. As seen in Figure 12.3, the density of vulnerabilities (w.r.t. BOC) in non-cloned blocks is much higher than that in code clones.

It is highly probable that a larger portion of source code contains more vulnerabilities than a smaller portion of source code, which might be a reason why non-cloned code seems to have more vulnerabilities as observed in Figure 12.2(a) and Figure 12.3. To verify this possibility, we compute the distribution of LOC in non-cloned code and different types of clones over all the systems as presented in Figures 12.4(a) and 12.4(b) respectively. In Figure 12.4(a), we find that the number of LOC in non-cloned code is significantly higher compared to cloned code. We also compute the lengths of non-cloned and cloned code blocks in terms of average LOC over all the systems that are found to be 48.69 and 12.97 respectively. Thus, the possibility of influence of code size (in terms of LOC) on the number and density of vulnerabilities w.r.t. *BOC* is found to be true.

We, therefore, continue our investigations at a deeper level using the densities of vulnerabilities w.r.t. *LOC*. The box-plot in Figure 12.5 presents the densities of vulnerabilities w.r.t. LOC (computed using Equation 12.3 and Equation 12.4) found in non-cloned code and in different types of clones over all the subject systems. Figure 12.5 shows that both the median and average of densities of vulnerabilities (w.r.t. LOC) in cloned code (all clones) are higher compared to non-cloned code.

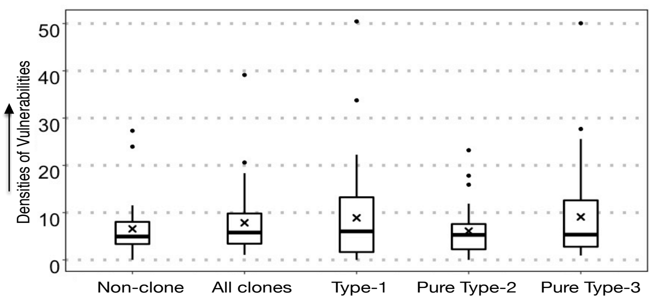


Figure 12.5: Densities of $\mathcal{FDV}$ w.r.t. LOC

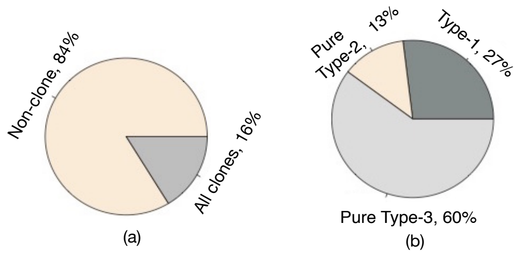


Figure 12.6: Distribution of $\mathcal{CDV}$ (a) in Cloned and Non-cloned Code and (b) in Different Types of Clones

We conduct a *MWW* test to measure the statistical significance of differences in the densities of vulnerabilities (w.r.t. LOC) in cloned and non-cloned code. The p-value ($p = 0.20, p > \alpha$) obtained from the *MWW* test implies that observed differences in the densities of vulnerabilities (w.r.t. LOC) in cloned and non-cloned code across all the subject systems are not statistically significant.

Analysis Using $\mathcal{CDV}$: Figures 12.6(a) and 12.6(b) present the distributions of $\mathcal{CDV}$ in non-cloned code and in different types of clones respectively over all the systems. Interestingly, the pattern of the distributions of vulnerabilities is similar to the pattern observed in Figures 12.2(a) and 12.2(b) drawn for $\mathcal{FDV}$.

Similar to Figure 12.5, Figure 12.7 presents the densities of vulnerabilities (i.e., $\mathcal{CDV}$) w.r.t. LOC (computed using Equation 3 and Equation 4) found in non-cloned code and in different types of clones over all the subject systems. As seen in Figure 12.7, the average of densities of vulnerabilities is higher in cloned code (all clones) compared to non-cloned code, although the median of densities of vulnerabilities is slightly higher in non-cloned code as opposed to code clones. Again, we conduct a *MWW* test to measure the statistical significance of differences in the densities of vulnerabilities (w.r.t. LOC) in cloned and non-cloned code. The p-value ($p = 0.42, p > \alpha$) obtained from the statistical test implies that differences in the densities of vulnerabilities (w.r.t. LOC) in cloned and non-cloned code across all the subject systems do not differ significantly. This result agrees with that obtained for $\mathcal{FDV}$. Therefore, we derive the answer to the *RQ1* as follows:

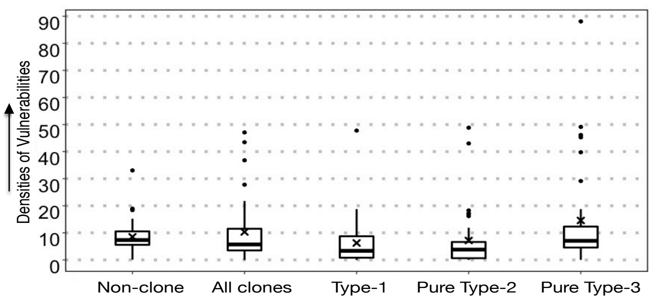


Figure 12.7: Densities of $\mathcal{CDV}$ w.r.t. LOC

Ans. to RQ1: *Densities of vulnerabilities in cloned code are NOT significantly higher than non-cloned code.*

12.4.2 Densities of Vulnerabilities in Different Types of Clones

Analysis Using $\mathcal{FDV}$: The distribution of $\mathcal{FDV}$ portrayed in Figure 12.2(b) shows that the *pure Type-2* clones are found to have the minimum vulnerabilities whereas the number of vulnerabilities found in *Type-1* clones is higher than that in pure *Type-2* clones. The vulnerabilities found in cloned portion of source code are found to be dominated by those found in *pure Type-3* clones. However, the majority of cloned LOC are also in *pure Type-3* clones as can be observed in Figure 12.4(b), which might be a reason why a higher number of vulnerabilities are found in code clones of this particular category.

As seen in Figure 12.5, the densities of vulnerabilities w.r.t. *LOC* in different categories of code clones follow the same pattern of densities of vulnerabilities w.r.t. *BOC* where *pure Type-3* clones show the highest average density of vulnerabilities followed by *Type-1* clones, and *pure Type-2* clones show the lowest average density. Although noticeable differences are observed in the *averages* of densities of vulnerabilities, we do not see much differences in the *medians*. To determine the statistical significance of our observations, we conduct a *Kruskal-Wallis* test between the densities of $\mathcal{FDV}$ (w.r.t. LOC) of the three categories of clones. The *p*-value ($p = 0.5901, p > \alpha$) obtained from the *Kruskal-Wallis* test suggests no significant differences in the distributions of densities of vulnerabilities.

Analysis Using $\mathcal{CDV}$: As observed in Figure 12.2(b) and Figure 12.6(b), the patterns of distributions of both $\mathcal{FDV}$ and $\mathcal{CDV}$ in different types of clones are very similar. However, a comparison of Figure 12.5 and Figure 12.7 makes some differences visible. In contrast with $\mathcal{FDV}$ (Figure 12.5), the average and median densities of $\mathcal{CDV}$ (Figure 12.7) in *Type-1* and *pure Type-2* code clones are almost equal while both the average and median are noticeably higher in *pure Type-3* clones. We also see that the *average* densities of both $\mathcal{FDV}$ and $\mathcal{CDV}$ are the highest in *pure Type-3* clones.

Again, to determine the significance of differences of densities of $\mathcal{CDV}$ in different types of clones, we conduct a *Kruskal-Wallis* test between the densities of $\mathcal{CDV}$ (w.r.t. LOC) of the three

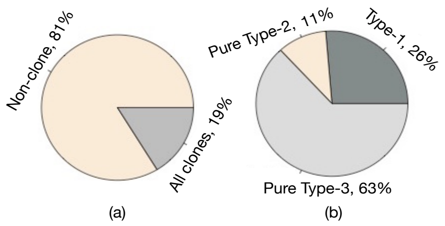


Figure 12.8: Cumulative Severity Scores of $\mathcal{FDV}$ (a) in Cloned and Non-cloned Code and (b) in Different Types of Clones

categories of clones. The p-value ($p = 0.008086, p < \alpha$) obtained from the *Kruskal-Wallis* test suggests significant differences in the distributions of densities of vulnerabilities. To determine the significance of the pairwise difference, we conduct a *Kruskalmc* post-hoc analysis. The test suggests statistical significance differences in the distributions of $\mathcal{CDV}$ in *pure Type-3* clones against *Type-1* and *pure Type-2* clones. The computed *Cliff's delta d* values 0.373 and 0.404 between *pure Type-3* and *Type-1* and between *pure Type-3* and *pure Type-2* respectively, indicate medium effect sizes. Based on our analyses of both $\mathcal{FDV}$ and $\mathcal{CDV}$, we now answer the *RQ2* as follows:

Ans. to RQ2: *Pure Type-3 clones are the most insecure category of clones, while Type-1 and pure Type-2 clones are almost equal in terms of security vulnerability.*

12.4.3 Severity of Security Risks in Cloned and Non-Cloned Code

Figures 12.8(a) and 12.8(b) present the distributions of cumulative severity scores of $\mathcal{FDV}$ in non-cloned code and in different types of clones respectively over all the systems. As seen in Figure 12.8(a), as much as 81% of total severity scores of $\mathcal{FDV}$ is associated with non-cloned code, which can be expected as non-cloned code contributes 93% of the entire source code (Figure 12.4(a)).

The box-plot in Figure 12.9 presents the risk severity scores per LOC (computed using Equation 12.5 and Equation 12.6) for non-cloned code and different types of clones for each of the subject systems. Figure 12.9 shows that both the median and average of the risk severity scores over all the systems in cloned code (all clones) are higher compared to non-cloned code. We conduct a *MWW* test to measure the statistical significance of these observed differences. The p-value ($p = 0.0494, p < \alpha$) obtained from the test indicates statistical significance of the differences. The computed *Cliff's delta d* value 0.381 suggests the effect size is medium. Therefore, we derive the answer to the RQ3 as follows:

Ans. to RQ3: *The security vulnerabilities in cloned code are significantly riskier than those in non-cloned code.*

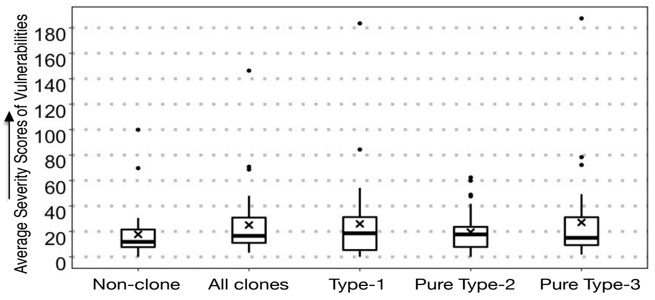


Figure 12.9: Risk Severity Scores per KLOC in Cloned and Non-cloned Code

12.4.4 Severity of Security Risks in Different Types of Clones

The distribution of cumulative severity scores of $\mathcal{FDV}$ in different types of code clones depicted in Figure 12.8(b) shows that vulnerabilities in *pure Type-2* clones have posed the lowest cumulative severity score whereas *pure Type-3* clones show the highest severity score and *Type-1* clones fit in between. Again, Figure 12.9 shows that the *average* severity scores of vulnerabilities in *pure Type-3* and *Type-1* clones are almost equal while *pure Type-2* clones have slightly lower severity score compared to the former two. Moreover, we do not see much differences in the *medians*.

To determine the statistical significance of our observations, we conduct a *Kruskal-Wallis* test between *average* severity scores of vulnerabilities (w.r.t. LOC) of the three categories of clones. The p-value ($p = 0.5901, p > \alpha$) obtained from the *Kruskal-Wallis* test suggests no significant differences in the distributions of severities of vulnerabilities. Based on the findings, we now answer the *RQ4* as follows:

Ans. to RQ4: *There is no significant difference in the severity of security vulnerabilities found in different types of code clones.*

12.4.5 Frequently Encountered Categories of Vulnerabilities

Analysis Using $\mathcal{FDV}$: For each CWE category of $\mathcal{FDV}$ identified in the subject systems, we compute the densities of those separately for cloned code and non-cloned code in each of the subject systems (using Equation 12.7). Then we distinguish five CWE categories, which have the highest vulnerability densities in cloned code over all the subject systems. Let $\mathcal{D}_c$ denote the set of these five CWE categories. Similarly, we form another set $\mathcal{D}_{\bar{c}}$ consisting of five CWE categories of vulnerabilities having the highest densities in non-cloned code. By the union of these two sets, we obtain a set $\mathcal{D}$ of top six CWE categories of vulnerabilities that have the highest densities across both cloned and non-cloned code. Mathematically, $\mathcal{D} = \mathcal{D}_c \cup \mathcal{D}_{\bar{c}}$.

Figure 12.10 presents the distributions of densities of these top six CWE categories of vulnerabilities in cloned and non-cloned code in each of the subject systems. As we see in Figure 12.10, the *average* densities of the CWE-807 category of vulnerabilities is higher in non-cloned code compared

Table 12.6: MWW tests over densities of top six CWE categories of $\mathcal{FDV}$ per KLOC in cloned and non-cloned code

Categories of $\mathcal{FDV}$	**P-Value**	**Significant?**	***Cliff's delta d***
CWE-78	0.0233	Yes ($p < \alpha$)	0.384 (medium)
CWE-120	0.3264	No ($p > \alpha$)	not applicable
CWE-134	0.2809	No ($p > \alpha$)	not applicable
CWE-190	0.4681	No ($p > \alpha$)	not applicable
CWE-362	0.4051	No ($p > \alpha$)	not applicable
CWE-807	0.0012	Yes ($p < \alpha$)	0.379 (medium)

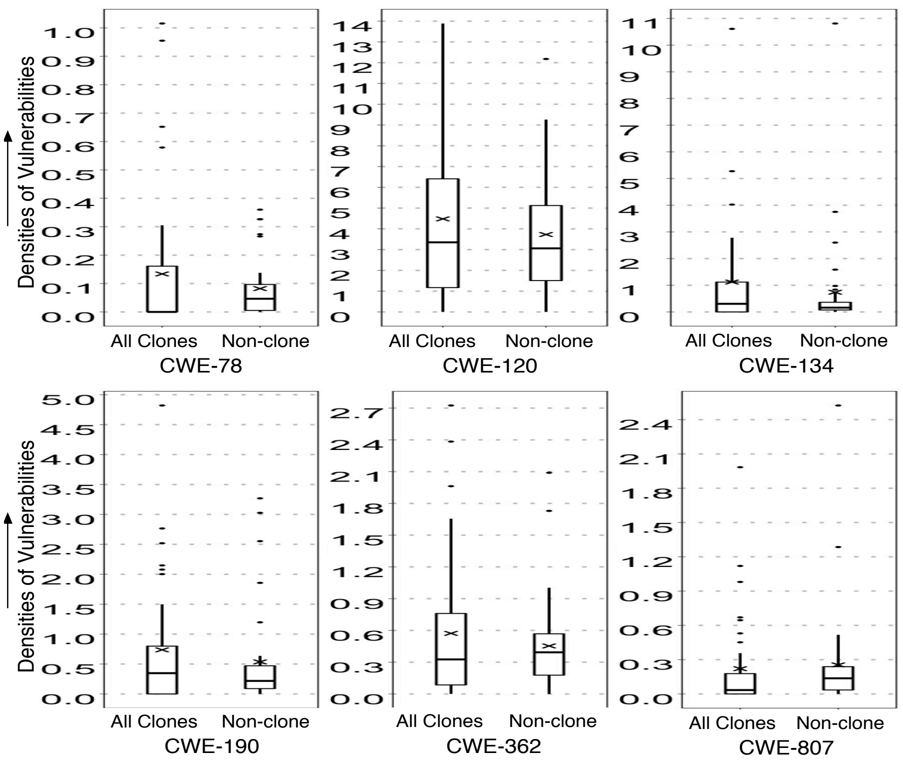


Figure 12.10: Densities of top six CWE categories of $\mathcal{FDV}$ per KLOC

to clones. The opposite is observed for the rest five CWE categories. For each of these six CWE categories, we separately conduct the *MWW* tests to determine the statistical significance of the differences in densities of vulnerabilities in cloned and non-cloned code. The resulted *p*-values of the tests presented in Table 12.6 indicate that for CWE-78 and CWE-807 the differences are statistically significant and otherwise for the rest four CWE categories. Then, we compute the *Cliff's delta d* values for the CWE-78 and CWE-807 categories and find medium effect sizes for both. Thus, we can distinguish the CWE-78 category vulnerabilities as a category of vulnerabilities, which appear in cloned code more frequently than in non-cloned clone. We can also distinguish the CWE-807 category vulnerabilities having the opposite characteristics.

Analysis Using $\mathcal{CDV}$: Using similar procedure followed for $\mathcal{FDV}$, we obtain the set $\mathcal{D}$ of five CWE categories of vulnerabilities from $\mathcal{CDV}$. Figure 12.11 presents the distribution of densities of

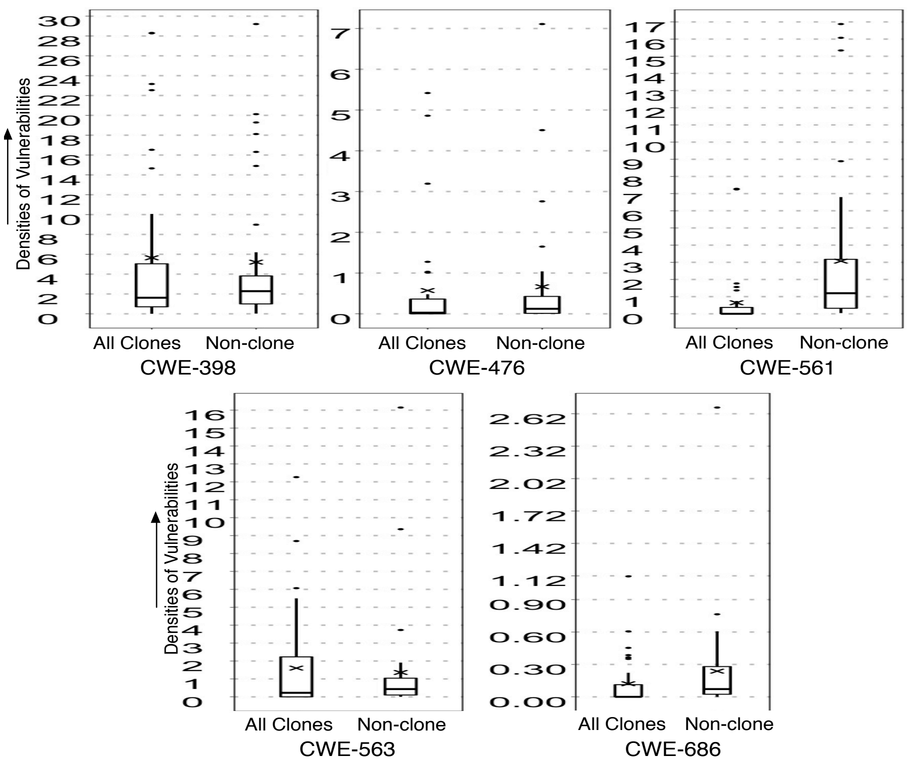


Figure 12.11: Densities of top five CWE categories of $\mathcal{CDV}$ per KLOC

Table 12.7: MWW tests over densities of top five CWE categories of $\mathcal{CDV}$ per KLOC in cloned and non-cloned code

Categories of $\mathcal{CDV}$	**P-Value**	**Significant?**	***Cliff's delta d***
CWE-398	0.2980	No ($p > \alpha$)	not applicable
CWE-476	0.0630	No ($p > \alpha$)	not applicable
CWE-561	0.0321	Yes ($p < \alpha$)	0.692 (large)
CWE-563	0.1538	No ($p > \alpha$)	not applicable
CWE-686	0.0012	Yes ($p < \alpha$)	0.372 (medium)

those five CWE categories of vulnerabilities over all the subject systems in cloned and none-cloned code. Again, for each of five CWE categories of vulnerabilities, we separately conduct *MWW* tests to determine the significance of differences in densities of $\mathcal{CDV}$ in clones and non-cloned code and also compute *Cliff's delta d* values for those distributions where $p < \alpha$.

The resulted p-values and computed effect sizes of those tests are presented in Table 12.7. Combining our observation in Figure 12.11, the p-values and the effect sizes in Table 12.7, we can infer that the densities of vulnerabilities of CWE-561 and CWE-686 categories are significantly higher with medium to large effect sizes in non-cloned code compared to cloned code. Thus, we have been able to distinguish two CWE categories of vulnerabilities whose appearances are dominant in non-cloned code. Based on our analysis of both $\mathcal{FDV}$ and $\mathcal{CDV}$, we derive the answer to the *RQ5* as follows:

Ans. to RQ5: *It is possible to distinguish particular CWE categories of vulnerabilities, which frequently appear in cloned code (or non-cloned code).*

12.5 Threats to Validity

Construct Validity: Although we have used two well-known tools for security vulnerability detection, we discarded from our study those reported vulnerabilities that the tools failed to associate with a CWE enumeration. We deliberately excluded those because a so-called vulnerability without an associated CWE number can actually be a stylistic issue that the tools erroneously report as a security vulnerability. In the selection of the two dominant sets of CWE categories of vulnerabilities (Section 12.4.5), we picked top five CWE categories of vulnerabilities for each set having the highest densities in cloned and non-cloned code. Although those chosen vulnerabilities cover more than 85% instances of vulnerabilities found in all the systems, this choice may still be considered as a threat to validity of this work. The computation of averages of the ordinal values of severity scores in the analyses for *RQ3* and *RQ4* can be questioned, although it serves our purpose of analyzing comparative severities of groups of vulnerabilities.

Internal Validity: While detecting vulnerabilities, false positives and false negatives could be two major threats to validity of this study. Hence, for vulnerability detection, we have used two different tools, `Flawfinder` [9] and `Cppcheck` [13]. `Flawfinder` is known to have high recall [292, 295, 297, 296], and `Cppcheck` is reputed for its high precision with zero false positives [13]. Moreover, while using `Flawfinder`, we used a customized configuration, which significantly minimize the false positives [299], as also demonstrated in Section 12.3.3 of this chapter. In addition, some types of vulnerabilities (e.g., *Null Pointer Dereference*) may span over multiple statements, which may partly overlap with both cloned and non-cloned code. Such a phenomenon couldn't be captured in the study as the vulnerability detection tools report only a line number in source file indicating the location of a particular vulnerability detected.

The clone detector, `NiCad` [138], used in our study, is reported to be very accurate in clone detection [138], and we have carefully set `NiCad`'s parameters for the detection of *Type-1*, *Type-2*, and *Type-3* clones. Moreover, we have taken care to avoid double-counting of nested blocks and vulnerabilities identified in them. In addition, we have manually verified the correctness of computations for all the metrics used in our work. Thus, we develop high confidence in the internal validity of this study.

External Validity: Although our study includes a large number of subject systems, all the systems are open-source and written in C. Thus the findings from this work may not be generalizable for industrial systems and source code written in languages other than C.

Reliability: The methodology of this study including the procedure for data collection and analysis is documented in this chapter. The subject systems being open-source, are freely accessible while the tools `Flawfinder`, `Cppcheck` and `NiCad` are also available online. Therefore, it should be possible to replicate the study.

12.6 Related Work

As mentioned before, no other work in the literature include a comparative study of security vulnerabilities in code clones and non-cloned source code. Hence, the studies which examined the characteristics and impacts of code clones are considered relevant to our work. Some studies included a comparative analysis of certain characteristics in clones against non-cloned code, as discussed below.

Lozano and Wermelinger [73] analyzed revisions of only four open-source projects and suggested that having a clone may increase the maintenance effort for changing a method. Hotta et al. [79] studied the changeability of cloned and non-cloned code in revisions of 15 open-source software systems. They reported code clones not to have any negative impact on software changeability, which contradicts the claim of Lozano and Wermelinger [73].

Towards understanding the stability of code clones, Lozano et al. [91] studied changes clones across revisions of only one software system and reported that a vast majority of methods experience larger and frequent changes when they contain cloned code. Based on a study on the revisions of 12 software systems, Mondal et al. [82] also reported code clones to be less stable. However, opposite results are reported from the other studies [283, 78, 90, 80]. In another study, Sajnani et al. [285] attempted to identify the relationships of code clones with statically identified bugs in systems written in Java. They found considerably lower number of bugs in code clones compared to non-cloned code.

Recently, Islam and Zibran [95] compared a large comparative study of the code ‘vulnerabilities’ in cloned and non-cloned code. In their work, ‘vulnerability’ was defined as the “problems in the source code identified based on bad coding patterns i.e, code smells". Our study is inspired from their work and significantly differs from theirs. First, in contrast with their study of code smells, we have studied the real security flaws widely known as security vulnerabilities. Second, they studied software systems written in Java, while we have studied systems written in C. Third, they used *PMD* [12] to detect code smells, whereas we have used two separate tools *Flawfinder* and *Cppcheck* to detect security vulnerabilities in source code. In addition, we have analyzed the severities of security vulnerabilities, while such severity of code smells was not studied in the aforementioned work of Islam and Zibran.

Attempts are also made to explore fault-proneness of clones by relating them with bug-fixing changes obtained from commit history. Such a study was conducted by Jingyue et al. [87], who reported that only 4% of the bugs were found in duplicated code. In a similar study, Rahman et al. [25] also observed that majority of bugs were not significantly associated with clones. Another study [301] along the same line reported that 55% of bugs in cloned code can be replicated bugs.

As discussed before, the contradictory results are often reported from comparative studies between clones and non-cloned code. This implies the necessity of further comparative investigations from a different dimension, which is exactly what we have done in this study. We have carried out a comparative investigation of *security vulnerabilities* in clones and non-coned code, which was missing in the literature. In addition, the comparative analysis of vulnerabilities in *Type-1*, *pure Type-2*, and *pure Type-3* clones is another important aspect of our work.

12.7 Summary

In this chapter, we have presented a large quantitative empirical study of the security vulnerabilities in clones and non-cloned code clones in 34 open-source software systems (8.7 million LOC) written in C. To the best of our knowledge, no such studies exists in the literature that performed a comparative analysis of security vulnerabilities in cloned and non-cloned code. For the detection of security vulnerabilities in source code, we have used two different tools (`Flawfinder` and `Cppcheck`), one of which is known to have high recall while the other is reputed for its high precision. For clone detection, we used a state-of-the-art clone detector, `NiCad`, which is also reported to have high accuracy.

Our study reveals that the security vulnerabilities found in code clones have higher severity of security risks compared to those in non-cloned code. However, the proportion (i.e., density) of vulnerabilities in clones and non-cloned code does not have any significant difference. Among the three categories (i.e., *Type-1*, *pure Type-2*, and *pure Type-3*) of code clones studied in our work, *pure Type-3* clones are found to be the most insecure whereas *Type-1* and *pure Type-2* clones are nearly equal in terms of the vulnerabilities found in them. The results are validated in the light of statistical significance.

The findings from this study advance our understanding of the characteristics, impacts, and implications of code clones in software systems. These findings will help in identifying problematic clones, which demand extra care and those vulnerabilities about which the developers need to be particularly cautious about while reusing code by cloning.

In future, we plan to conduct qualitative analyses to advance our understanding of such results. Moreover, we plan to perform similar analyses using software systems written in languages other than C to verify to what extent the findings from this study also applies to a broader range of source code written in diverse programming languages.

Chapter 13

Characteristics of Buggy Code Clones

In the last chapter (Chapter 12), we presented a study to explore and understand the security vulnerabilities and their severity in different types of clones compared to non-clone code. We know that code clone is an immensely studied code smell. However, not all the clones in a software system are equally harmful. This chapter presents a comparative study to distinguish the characteristics (from a code quality perspective) of buggy and non-buggy clones.

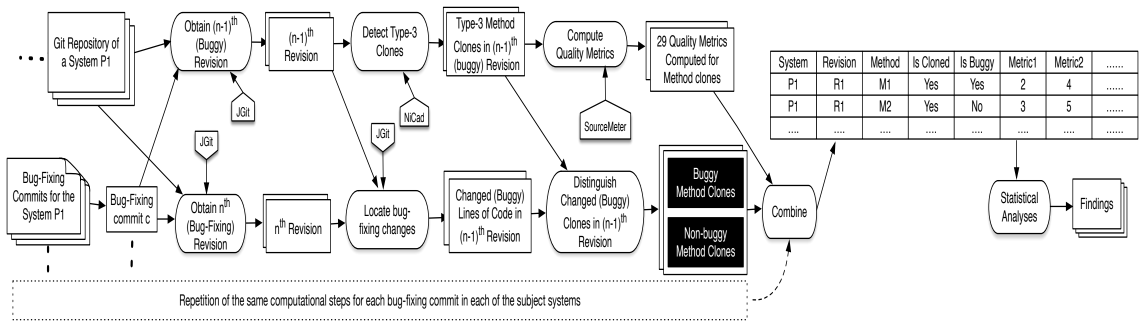
The rest of this chapter is organized as follows. In Section 13.1, we introduce the motivation of the work. In Section 13.2, we describe the setup and procedure of our empirical study. Section 13.3 presents our analysis and the findings derived from this study. In Section 13.4, we discuss the possible threats to the validity of our study. Section 13.5 includes related work, and Section 13.6 concludes the chapter.

13.1 Introduction

In the past, several studies examined the comparative stability of clones as opposed to non-cloned code [283, 78, 79, 80, 81, 82, 302], comparative vulnerabilities in cloned and non-cloned code [95, 96], relationships of clones with bug-fixing changes [83, 84, 85, 72, 86, 87, 25, 88, 89], change-proneness of clones [303], and the impacts of clones on program's changeability [90, 73, 91]. There have also been studies on clone removal in program history [289, 290, 291]. Existing literature suggest that, (i) clones are problematic in many cases, (ii) not all clones are equally harmful [71], and (iii) it is not practically feasible to remove *all* clones from a system through aggressive refactoring [286, 304, 75]. Therefore, for cost-effective clone management, we must distinguish the characteristics of clones, which make them problematic (e.g., buggy).

Not much work is done along this direction. Majority of earlier work made comparisons between clones and non-cloned code with respect to certain criteria (e.g., stability, vulnerability, bug-proneness). Only a few earlier studies suggested merely a handful of characteristics that might make certain clones detrimental. Such characteristics include late-propagation of clones [283, 83] and stability/change-proneness of clones [302, 303].

In this chapter, using 29 code quality metrics, we study the characteristics of buggy and non-buggy clones aiming to identify certain quality metrics, which can indicate potential bug-proneness of clones. In particular, we address the following four research questions.


Git Repository of a System P1
Obtain (n-1)th (Buggy) Revision
JGit
(n-1)th Revision
Detect Type-3 Clones
NiCad
Type-3 Method Clones in (n-1)th (buggy) Revision
Compute Quality Metrics
SourceMeter
29 Quality Metrics Computed for Method clones
Bug-Fixing Commits for the System P1
Bug-Fixing commit c
JGit
Obtain nth (Bug-Fixing) Revision
nth Revision
JGit
Locate bug-fixing changes
Changed (Buggy) Lines of Code in (n-1)th Revision
Distinguish Changed (Buggy) Clones in (n-1)th Revision
Buggy Method Clones
Non-buggy Method Clones
Combine
Statistical Analyses
Findings
Repetition of the same computational steps for each bug-fixing commit in each of the subject systems

System	Revision	Method	Is Cloned	Is Buggy	Metric1	Metric2	......
P1	R1	M1	Yes	Yes	2	4	......
P1	R1	M2	Yes	No	3	5	......
....	....	....	....	....	....	....	......

Figure 13.1: Procedural Steps of the Empirical Study

RQ1: *Are buggy cloned methods more complex than non-buggy cloned methods or vice versa*? —Failure-prone software entities are statistically correlated with code complexity measures [305]. However, there is no single set of complexity metrics that can indicate defect [305]. Here, we investigate 15 complexity metrics to statistically measure their dominance in buggy and non-buggy cloned code.

RQ2: *Are buggy cloned methods larger than non-buggy cloned methods or vice-versa?* — Earlier studies [306, 68] found cloned methods to be smaller than the non-cloned methods. However, it is unknown whether there is any substantial size difference between buggy and non-buggy clones. Using seven metrics, we investigate the size difference between buggy and non-buggy cloned methods.

RQ3: *Are buggy cloned methods more documented than non-buggy cloned methods or vice-versa?* — Its is suggested that when cloning a piece of code the variations should be well documented in order to facilitate bug fix propagation [71]. As such, undocumented or poorly documented clones can have high possibility of causing bugs. Hence, we investigate this possibility by analyzing five source code documentation metrics.

RQ4: *Are buggy cloned methods more coupled than non-buggy cloned methods?* — Low coupling and high cohesion are two prominent software design qualities expected to keep a software system's inherent complexity manageable. In other words, highly coupled code is harder to manage and could be bug-prone. Thus, we investigate whether coupling metrics' values are significantly higher in buggy cloned code compared to non-buggy cloned code.

13.2 Study Setup

The procedural steps of our empirical study are summarized in Figure 13.1, and described in the following subsections.

13.2.1 Subject Systems

We study 2,077 revisions of three open-source software systems written in Java. These subject systems, as listed in Table 14.1, are available at the GitHub repository. In Table 14.1, we present the total number of revisions and number of bug-fixing revisions of each subject system along with the number of source lines of code (LOC) in the last revision. We choose these three subject systems as these systems have variations in application domains, sizes, number of revisions, and are also used in another study [232].

Table 13.1: Subject Systems

Subject System	**Application Domain**	**LOC (last rev.)**	**Total # of Revisions**	**# of Bug-Fixing Rev.**
Netty	Network	1,078,493	8,534	1,103
Presto	SQL	2,869,799	11,909	841
Facebook-android-SDK	Social Networking	172,695	671	133
Total over all the systems		4,120,987	21,114	2,077

13.2.2 Clone Detection

Code clones at different levels of syntactic similarities appear in source code. Identical pieces of source code with or without variations in whitespaces (i.e., layout) and comments are called *Type-1* clones [286]. *Type-2* clones are syntactically identical code fragments with variations in the names of identifiers, literals, types, layout and comments [286]. Code fragments, which exhibit similarities as of *Type-2* clones and also allow further differences such as additions, deletions or modifications of statements are known as *Type-3* clones [286].

By definition, *Type-3* clones include both *Type-1* and *Type-2*. In this work, we study *Type-3* clones at the granularity of method bodies. We use the `NiCad` [138] clone detector (version 3.5), to detect method/function clones having at least five LOC. In detection clones, 'blind renaming' option of `NiCad` is kept enabled and UPIT (i.e., dissimilarity threshold) is set to 0.3. Further details on `NiCad`'s tuning parameters and their influences on clone detection can be found elsewhere [138].

13.2.3 Distinguishing Buggy Clones

Consider a bug-fixing commit c resulting in the n^{th} revision of a system. If a particular method m is modified in the bug-fixing commit c, then it implies that the modification is necessary to fix the bug. Thus, the method m in the $(n-1)^{\text{th}}$ revision is considered a buggy method. In this work, we use the bug-fixing commits identified by Ray et al. [232]. These bug-fixing commits are distinguished through matching keywords (e.g., bug, defect, issue) in the commit messages and are reported to be 96% accurate [232].

For a project $\mathcal{P}$, we determine the buggy cloned methods by performing the following steps: (i) we collect all the bug-fixing commits for $\mathcal{P}$ from the dataset of Ray et al. [232]. (ii) For each bug-fixing commit c, using `JGit` [307], we obtain the n^{th} revision of $\mathcal{P}$, which is the result of the commit c. (iii) Then, we obtain the $(n-1)^{\text{th}}$ revision of $\mathcal{P}$. (iv) The changes between the n^{th} and $(n-1)^{\text{th}}$ revisions of $\mathcal{P}$ are captured using `JGit` along with changed lines' numbers in the java source files. (v) Using `NiCad` [138], we detect *Type-3* method clones in the $(n-1)^{\text{th}}$ revisions of $\mathcal{P}$ and record their locations and boundaries (start and end line numbers in source files). (vi) Whether a commit c affected a cloned method is determined by checking if any of changed lines' number identified in the step-iv falls within the boundary of the clone. (vii) Clones that are affected by the bug-fixing commits are identified as buggy clones while the rest other clones are considered non-buggy.

Several other studies [301, 308, 303, 25] also adopted similar approaches for distinguishing buggy source code.

13.2.4 Computation of Source Code Quality Metrics

We use total 29 source code quality metrics grouped into four categories- (i) *complexity metrics*, which measure the complexity of source code elements; (ii) *size metrics*, which measure the basic properties of the analyzed system in terms of different cardinalities (e.g., number of code lines). (iii) *documentation metrics*, which measure the amount of comments and documentation of source code elements in the system; and (iv) *coupling metrics*, which measure the interdependencies of source code elements. All the metrics are listed and briefly described in Table 13.2. Detail descriptions of those metrics can be found in the user manual of `SourceMeter` [139], which is a proprietary static source code analyzer tool we use in this work. We opt out of describing the metrics in details due to space limitations.

Using a free version of `SourceMeter` [139] (version 8.2.0-x64-linux), we compute all the metrics in Table 13.2 for each of the buggy and non-buggy clones. The tool's configuration parameters are kept at their defaults.

13.3 Analysis and Findings

We carry out our analyses in the light of each of the 29 quality metrics separately averaged for buggy and non-buggy clones. For testing statistical significance we apply *Mann-Whitney-Wilcoxon (MWW)* test [234] with $\alpha = 0.05$. To measure effect size, we compute *Cliff's delta d* [234].

Both of these non-parametric statistical tests do not require normal distribution of data, and thus suit well for our purpose. We consider *significant difference* exists between distributions if p-value of a *MWW* test is found to be less than α and *Cliff's delta d* value is not negligible (i.e., $|d| > 0.15$). In drawing the box-plots for our analyses, we normalize the metrics values using the widely used *Min-Max* [309] method.

Table 13.2: Source Code Quality Metrics Used in this Study

Category	**Metric**	**Description**
Complexity metrics	↓ HCPL	Hal. Calculated Program Length
	↓ HDIF	Hal. Difficulty
	↓ HEFF	Hal. Effort
	↓ HNDB	Hal. Number of Delivered Bugs
	↓ HPL	Hal. Program Length
	↓ HPV	Hal. Program Vocabulary
	↓ HTRP	Hal. Time Required to Program
	↓ HVOL	Hal. Volume
	↑ MIMS	Maintainability Index (MS)
	↑ MI	Maintainability Index (OV)
	↑ MISEI	Maintainability Index (SEIV)
	↑ MISM	Maintainability Index (SV)
	↓ McCC	McCabe's Cyclomatic Complexity
	↓ NL	Nesting Level
	↓ NLE	Nesting Level Else-If
Size metrics	↓ LOC	Lines of Code
	↓ LLOC	Logical Lines of Code
	↓ NUMPAR	Number of Parameter
	↓ NOS	Number of Statements
	↓ TLOC	Total Lines of Code
	↓ TLLOC	Total Logical Lines of Code
	↓ TNOS	Total Number of Statements
Documentation metrics	↑ CD	Comment Density
	↑ CLOC	Comment Lines of Code
	↑ DLOC	Documentation Lines of Code
	↑ TCD	Total Comment Density
	↑ TCLOC	Total Comment Lines of Code
Coupling metrics	↓ NII	Number of Incoming Invocations
	↓ NOI	Number of Outgoing Invocations

Hal.=Halstead; MS= Microsoft version; OV=Original version
SEIV=SEI version; SV=SourceMeter version.
↑ by a metric indicates the higher the better for that metric
↓ by a metric indicates the lower the better for that metric.

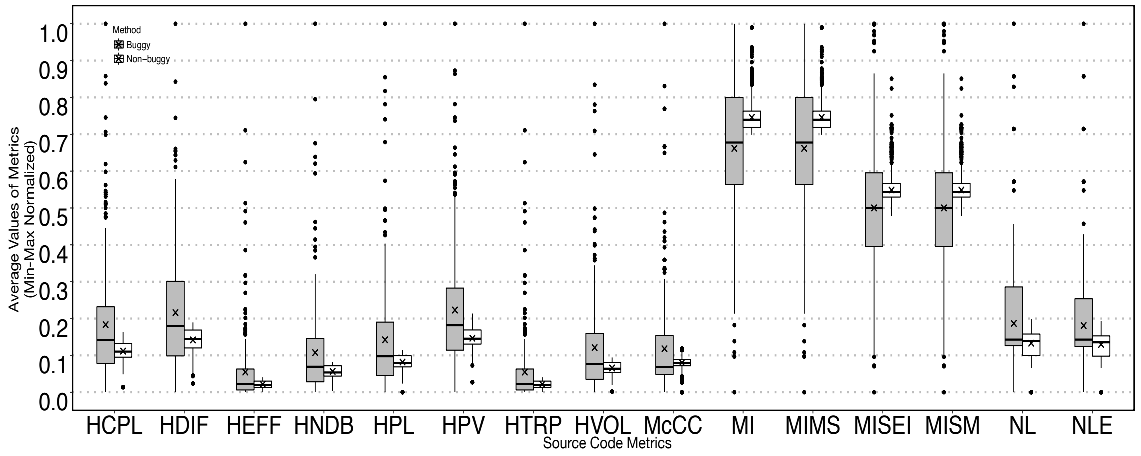


Figure 13.2: Distribution of Complexity Metrics' Values in Buggy and Non-buggy Cloned Code

13.3.1 Complexity of Buggy and Non-buggy Clones

In the box plot of Figure 13.2, we present the distribution of the average complexity metrics' values for buggy (grey boxes) and non-buggy (white boxes) clones for each studied revision of the subject systems. The 'x' marks in the box plots indicate the means over all the revisions across the subject systems.

As seen in Figure 13.2, for buggy clones, there are more variations in values of all the complexity metrics compared to those for non-buggy clones. Most importantly, considering the averages (marked with 'x'), all the 15 complexity metrics' values are worse for buggy clones. Buggy clones exhibit lower values for the four *maintenance index* related complexity metrics (i.e., MI, MIMS, MISEI, and MISM). For these four metrics higher values are desirable as indicated in Table 13.2. For the rest 11 complexity metrics, buggy clones are found to have higher values while lower values are desirable for these 11 metrics. These observations indicate that buggy clones are more complex and less maintainable compared to non-buggy clones.

To determine whether our observations are statistically significant, for each of the 15 complexity metrics, we separately conduct a one-sided pair-wise *MWW* test between the metric's values computed for buggy and non-buggy clones. The *p*-values obtained from these tests are presented in Table 13.3. The measurements of effect sizes (i.e., *Cliff's delta d*) corresponding to the MWW tests are also included in Table 13.3.

As seen in Table 13.3, the *p*-values obtained from MWW tests are less than α for all the complexity metrics except for three (HEFF, HTRP, and McCC). However, for two (HNDB and HVOL) of these 12 complexity metrics, the effect sizes computed in *Cliff's delta d* are found to be negligible. For rest of the 10 complexity metrics, the MWW tests indicate statistical significance in the

Table 13.3: *MWW* Tests over the Distribution of *Complexity* Metrics for Buggy and Non-buggy Clones

Source Code Metrics	P**-value**	**Cliff's delta** d	**Significant?**
HCPL	7.011×10^{-12}	0.2297 (small)	Yes
HDIF	2.23×10^{-11}	0.2239 (small)	Yes
HEFF	0.2787	Not applicable	No
HNDB	6.017×10^{-05}	0.1307 (negligible)	No
HPL	1.23×10^{-06}	0.1601 (small)	Yes
HPV	9.75×10^{-14}	0.2499 (small)	Yes
HTRP	0.5575	Not applicable	No
HVOL	2.75×10^{-05}	0.1371 (negligible)	No
MI	2.20×10^{-16}	-0.2882 (small)	Yes
MIMS	2.20×10^{-16}	-0.2882 (small)	Yes
MISEI	1.48×10^{-14}	-0.2614 (small)	Yes
MISM	1.48×10^{-14}	-0.2614 (small)	Yes
McCC	0.1872	Not applicable	No
NL	4.17×10^{-07}	0.1675 (small)	Yes
NLE	1.65×10^{-10}	0.2136 (small)	Yes

differences of the metrics' values for buggy and non-buggy clones while the *Cliff's delta d* values also suggest that the effect sizes are not negligible.

Hence, from our observations and statistical tests, we now derive the answer to *RQ1* as follows:

Ans. to RQ1: *Buggy clones have significantly higher complexity and lower maintainability compared to non-buggy code clones.*

13.3.2 Size Difference of Buggy and Non-buggy Clones

In Figure 13.3, we plot the distribution of the seven size metrics' values computed for buggy and non-buggy clones. Similar to the case of complexity metrics (discussed in Section 13.3.1), we see that the variations in all the size metrics is higher in buggy clones than in non-buggy ones.

Table 13.4: *MWW* Tests over Average Values of *Size* Metrics in Buggy and Non-buggy Cloned Code

Source Code Metrics	P**-Value**	**Cliff's delta** d	**Significant?**
LLOC	1.34×10^{-10}	0.2147 (small)	Yes
LOC	1.47×10^{-11}	0.2260 (small)	Yes
NOS	5.48×10^{-06}	0.1500 (small)	Yes
NUMPAR	3.44×10^{-06}	-0.1529 (small)	Yes
TLLOC	8.88×10^{-11}	0.2169 (small)	Yes
TLOC	1.12×10^{-11}	0.2275 (small)	Yes
TNOS	1.94×10^{-06}	0.1570 (small)	Yes

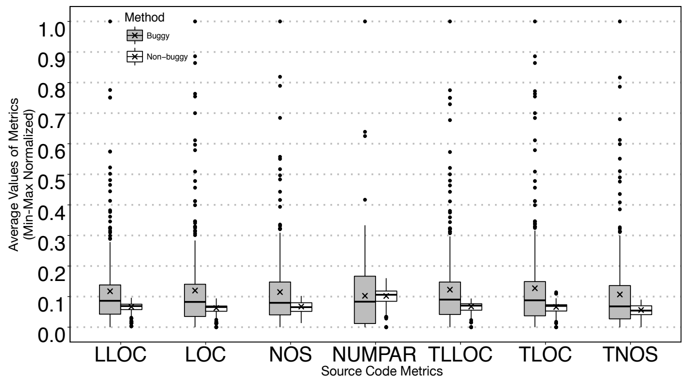


Figure 13.3: Size Metrics' Values in Buggy and Non-buggy Clones

Average values of all the size metrics are higher (i.e., worse) for buggy method clones except for NUMPAR (i.e., number of parameters). Surprisingly, the average value of this particular metric appear to be slightly higher for non-buggy clones. Again, to verify the significance of our observations and effect size, we conduct one-sided pair-wise *MWW* test and *Cliff's delta d* for each of the seven size metrics between their values computed for buggy and non-buggy clones. The results of the tests are presented in Table 13.4. The results in Table 13.4 suggest statistical significance (with non-negligible effect size) in the differences of all the seven size metrics' values computed for buggy and non-buggy clones. Based on the findings, we now answer the *RQ2* as follows:

Ans. to RQ2: *Compared to non-buggy clones, the buggy method clones have significantly higher code size measured in terms of the number of lines and statements. Surprisingly, non-buggy cloned methods are found to have higher number of parameters.*

13.3.3 Documentation in Buggy and Non-buggy Clones

Figure 13.4 depicts the distribution of average values of the five documentation metrics for buggy and non-buggy clones. As seen in the figure, the medians of all the documentation metrics' values are consistently *higher* (better) for non-buggy clones. On the contrary, for non-buggy clones, the mean values are *lower* (worse) for all the metrics except for DLOC (Documentation Lines of Code).

Table 13.5: *MWW* Tests over Average Values of *Documentation* Metrics in Buggy and Non-buggy Cloned Code

Source Code Metrics	***P*-Value**	**Cliff's delta *d***	**Significant?**
CD	2.20×10^{-16}	-0.2809 (small)	Yes
CLOC	3.68×10^{-16}	-0.277 (small)	Yes
DLOC	2.20×10^{-16}	-0.7243 (**large**)	Yes
TCD	9.58×10^{-16}	-0.2730 (small)	Yes
TCLOC	3.12×10^{-15}	-0.2680 (small)	Yes

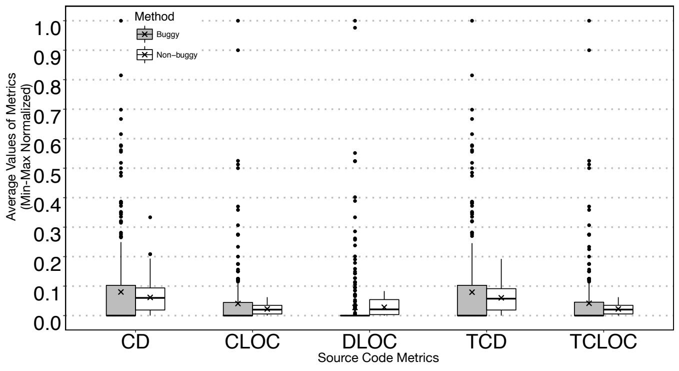


Figure 13.4: Documentation Metrics' Values in Buggy and Non-buggy Clones

Now, for the distributions of each of the five documentation metrics computed for buggy and non-buggy clones, we separately conduct a one-sided pair-wise *MWW* test and also measure effect size using *Cliff's delta d* to test our hypothesis that buggy clones have inferior documentation (i.e., lower documentation metrics value) than the non-buggy clones. The obtained *p*-values and *Cliff's delta d* values are presented in Table 13.5. As we see in Table 13.5, for all the five documentation metrics, the *p*-values indicate statistical significance and *Cliff's delta d* values suggest non-negligible effect sizes. Thus the statistical tests fail to reject our hypotheses for each of the documentation metrics. We, therefore, answer the research question *RQ3* as follows:

Ans. to RQ3: *The quality of documentation in buggy clones is significantly inferior to that in non-buggy clones.*

13.3.4 Coupling in Buggy and Non-buggy Clones

In Figure 13.5, we plot the distribution of average values of the two coupling metrics for buggy and non-buggy clones. As we see in Figure 13.5, for both the coupling metrics, the variations in buggy clones are higher compared to non-buggy clones. Again, for both the metrics, averages are higher for buggy clones, and the difference is more substantial for the NOI (Number of Outgoing Invocations) metric. The median for NOI is higher for buggy clones while for the NII (Number of Incoming Invocations) metric the medians are nearly equal for both buggy and non-buggy clones.

Table 13.6: *MWW* Tests over Average Values of *Coupling* Metrics in Buggy and Non-buggy Method Clones

Source Code Metrics	***P*-Value**	**Cliff's delta *d***	**Significant?**
NII	0.006414	−0.0926 (negligible)	No
NOI	2.20×10^{-16}	0.2972 (small)	Yes

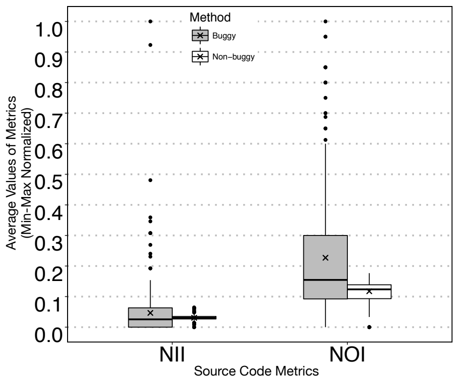


Figure 13.5: Coupling Metrics' Values in Buggy and Non-buggy Clones

Similar to previous analyses, we conduct one-sided pair-wise *MWW* test and compute *Cliff's delta d* separately for each of the coupling metrics to test the hypothesis that buggy clones are highly coupled (i.e., have higher coupling metrics' values) than the non-buggy clones. The results of the tests are presented in Table 13.6. As seen in the table, the *p*-values of MWW tests for both the metrics indicate statistical significance with $p < \alpha$. However, the *Cliff's delta d* values indicate non-negligible effect size for NOI, but negligible effect size for the NII metric.

In combination of the two metrics, we can say that buggy method clones are comparatively more coupled than non-buggy clones. Hence, we derive the answer to the research question *RQ4* as follows:

Ans. to RQ4: *Buggy method clones have significantly higher number of outgoing dependencies (i.e., outgoing invocations) compared to non-buggy method clones. However, there is no significant differences in the incoming dependencies (i.e., incoming invocations) of buggy and non-buggy cloned methods. Overall, buggy clones are more coupled than non-buggy clones.*

13.4 Threats to Validity

Construct Validity: We use the bug-fixing commits that are identified by Ray et al. [232]. These bug-fixing commits are distinguished based on matching of keywords (e.g., bug-fix, bug, issue, bug id) in the commit messages. There is a possibility that some portions of the bug-fixing commits could be regular commits (e.g., relevant to new feature implementations and improvements) and might not be genuinely relevant to bug-fix. However, this dataset of bug-fixing commits is reported to be 96% accurate [232]. In the identification of bug-fixing changes affected by a bug-fixing commit, there is a possibility that not all the affected lines of code are indeed responsible for the bug fixed in the bug-fixing commit. However, for this purpose, this is an acceptable approach also widely adopted in other studies [301, 308, 303, 25].

When a cloned method m is affected by bug-fixing changes in revision n, the method m is considered buggy at revisions $n - 1$. For rest of the revisions the method m is considered non-buggy. The content of the method m can safely considered non-buggy in any later revision ρ with $\rho > n$ since the bug is fixed in revision n. However, prior to the bug-fixing revision n, the method m may or may not be buggy, but in our work, we considered it non-buggy. This assumption can be argued as a threat to the construct validity of this work.

Internal Validity: In the detection of clones, we have used the `NiCad` clone detector, which is reported to be highly accurate [138, 310] and is widely used in many studies [301, 302, 68, 75]. The library `JGit` used in our work for locating changes between two revisions is also reliably used for similar purposes in other studies [311, 312]. Moreover, we manually verified the correctness of the computations probing with random samples. Thus, we develop high confidence in the internal validity of this study.

External Validity: Although our study includes a large number of revisions of three subject systems, all the systems are open-source and written in Java. Thus the findings from this work may not be generalizable for industrial systems and source code written in languages other than Java.

Reliability: The methodology of this study including the procedure for data collection and analysis is documented in this chapter. The subject systems being open-source, are freely accessible while the tool `NiCad` and library `JGit` are available online. All the bug-fixing commits are also available [232]. Therefore, it should be possible to replicate this study.

13.5 Related Work

Several attempts are made to explore fault-proneness of clones by relating them with bug-fixing changes obtained from commit history. Very recently, Mondal et al. [303] claimed that code clones, which were recently changed or created had high possibilities of containing bugs. Again, Rahman and Roy [302] found that stability and bug-proneness of code clones are related. The studies of Mondal et al. and Rahman and Roy only focused on recency of code changes and stability of cloned code respectively. On the other hand, our work, for the first time has investigated a comprehensive set of source code metrics to identify their relationships with bug-proneness of cloned code.

Selim et al. [88] investigated bug-proneness of code clones by combining source code and clone related metrics. However, they used only four source code metrics (e.g., Lines of Code, number of tokens, Nesting levels and Cyclomatic Complexity). Moreover, their investigation was based only on *Type-1* and *Type-2* method clones, thus missed out *Type-3* clones. In contrast to their work, we have used *Type-3* clones for this study that also includes *Type-1* and *Type-2* clones. Moreover, we have used 29 source code metrics to conduct the analyses in this study.

Wang et al. [313] related eight source code metrics with harmfulness of cloned code. They defined harmfulness of a piece of clone code based on maintenance cost. The higher the maintenance cost, the higher the clone code is harmful. In contrast, we have studied real buggy clones identified thorough bug-fixing changes and examined them using 29 software metrics.

Juergens et al. [72] studied inconsistent clones to relate with bugs. They used manual investigation to identify bugs in inconsistent clones, and concluded that unintentionally made inconsistent clones are more likely to contain defects. They hadn't conducted any statistical tests of significance of their finding. In contrast to their approach, we have mined source code repositories to identify bugs and conducted statistical tests of significance for all of our findings.

Rahman et al. [25] compared bug-proneness of clone and non-clone code and found non-clone code to be more bug-prone than clone code. However, their investigation was based on monthly snapshots of their subject systems, and thus, they had the possibility of missing buggy commits. In our study, we consider all the bug-fix revisions for each of the subject systems, thus, we have included all bug-fix commits.

Barbour et al. [83] suggested that late propagations due to inconsistent changes are prone to introduce software defects. While Lozano and Wermelinger [73] suggested that having a clone may increase the maintenance effort for changing a method, Hotta et al. [79] reported code clones not to have any negative impact on software changeability. Lozano et al. [91] reported that a vast majority of methods experience larger and frequent changes when they contain cloned code. Mondal et al. [82] also reported code clones to be less stable. However, opposite results are found from several other studies [283, 90, 78, 80].

Recently, Islam et al. [96] conducted a comparative study of *security vulnerabilities* in cloned and non-cloned code. Earlier Islam and Zibran [95] and Sajnani et al. [285] conducted two comparative studies of *code smells* in cloned and non-cloned code. However, they defined *code smells* as *vulnerability* and *bug patterns* in their respective studies. By targeting *code stability* and *dispersion of code changes* Mondal et al. also performed comparative studies in cloned and non-cloned code [314, 315].

Along the comparative studies, Saini et al. [306] conducted a study to compare source code quality metrics for cloned and non-cloned code. The study used 27 software quality metrics, categorized in three groups e.g., complexity, modularity, and documentation (code-comments). They did not find any statistically significant difference between the quality of cloned and non cloned methods for most of the metrics. However, we have compared the significance of differences of 29 software quality metrics' (categorized in four groups) values in buggy and non-buggy cloned methods instead of comparing them in cloned and non-cloned code.

In another study, Islam et al. [308] found the percentage of changed files due to bug-fix commits is significantly higher in clone code compared with non-clone code. They also found the possibility of severe bugs occurring is higher in clone code than in non-clone code. While all these above studies provide valuable insights about the characteristics of code clones, the clone literature lacked the comparative study of buggy and non-buggy cloned code characteristics in terms of source code quality metrics. This study has filled that gap in some extent and identify which source code quality metrics have contributed to buggy cloned code.

13.6 Summary

In this chapter, we have presented a comparative study of buggy and non-buggy *Type-3* method clones in terms of 29 code quality metrics grouped into complexity metrics, size metrics, documentation metrics, and coupling metrics. Our quantitative study is based on 2,077 bug-fixing revisions of three open-source software systems written in Java.

In our study, we have found that buggy clones have higher complexity and lower maintainability compared to non-buggy cloned methods. Moreover, buggy clones are found to be larger in size measured in terms of the number statements and lines of code. Surprisingly, we have found that the non-buggy method clones have higher number of parameters while too many parameters in functions are generally considered problematic and recognized as a code smell [74].

As expected, compared to non-buggy clones, documentation quality of buggy clones are found inferior. Overall, buggy clones are found to be more coupled than non-buggy clones. However, it is interesting to have found that the buggy cloned methods have significantly higher outgoing dependencies (i.e., outgoing invocations) compared to non-buggy clones. In case of incoming dependencies, no significant difference is found between buggy and non-buggy clones.

The results are validated with statistical tests of significance. However, there is a need for qualitative analyses to draw further insights into the reasons of our findings, especially for case of higher method parameters in non-buggy method clones and the for the case of higher outgoing dependencies in buggy clones. We plan to extend this work with qualitative analyses along with higher number of subject systems and their revisions.

The findings from this study advance our understanding of the characteristics and impacts of buggy clones on code quality. It appears that some of the code quality metrics can be good indicators for potentially buggy clones, and thus can be applied for identifying problematic clones for removal by refactoring or other especial treatments. For doing such, we need to derive a thresholding mechanism, which would indicate at what values of the quality metrics certain clones would be reported as potentially harmful. These remain within our plans for expanding this work.

Chapter 14

Exposing Bug-fix Patterns

From Chapter 4 to Chapter 13, we presented our studies to address the sub-problem I (i.e., detecting developers' sentiments/emotions), sub-problem II (i.e., understanding developers' sentiments),and sub-problem III (understanding impacts of developers copy-paste action). In this chapter, we conduct an in-depth quantitative and qualitative analysis over buggy revisions of software systems to identify the patterns of bug-fixings to address the sub-problem IV (i.e., understanding fixing patterns of developers mistakes that cause software bugs).

The remainder of this chapter is organized as follows. In Section 14.1, we introduce the motivation of the work. In Section 14.2, we describe the methodology of this study including short descriptions of the dataset and tools used. In Section 14.3, we identify dominant bug-fixing *edit* patterns. In Section 14.4, we derive the nesting patterns by distinguishing those nested code structures that host frequent bug-fix edits. In Section 14.5, we describe possible limitations and the threats to the validity of this work. Related work is discussed in Section 14.6. Finally, Section 14.7 concludes the chapter.

14.1 Introduction

Bug-fixing efforts consume a vast amount of total expenses in software maintenance [98] while nearly 80% of software cost is spent in maintenance [76]. Typical bug-fixing efforts mainly involve two types of tasks: (a) localization of bugs in source code, and (b) bug-fixing edits to source code. A deep understanding of the common bug-fixing patterns can immensely help in minimizing efforts in both of these tasks and also contribute to devising techniques for automated program repair (APR). A good understanding of the bug patterns can also help a developer to proactively avoid writing code that leads to program faults.

Bug-fixing efforts require a good understanding of the source code, intended edits, and their potential impacts. Studies [316, 317] find that code changes are repetitive in nature within and across code bases. Hence, mining code changes has become an effective way for program comprehension and deriving patterns of diverse categories including bug-fix patterns.

Early efforts in discovering bug-fix patterns highly depended on manual efforts [316, 318] in the analysis of textual differences among different program entities. However, manual effort is criticized for being error-prone, tedious, incomplete, and imprecise [319, 320, 321]. Recent efforts made

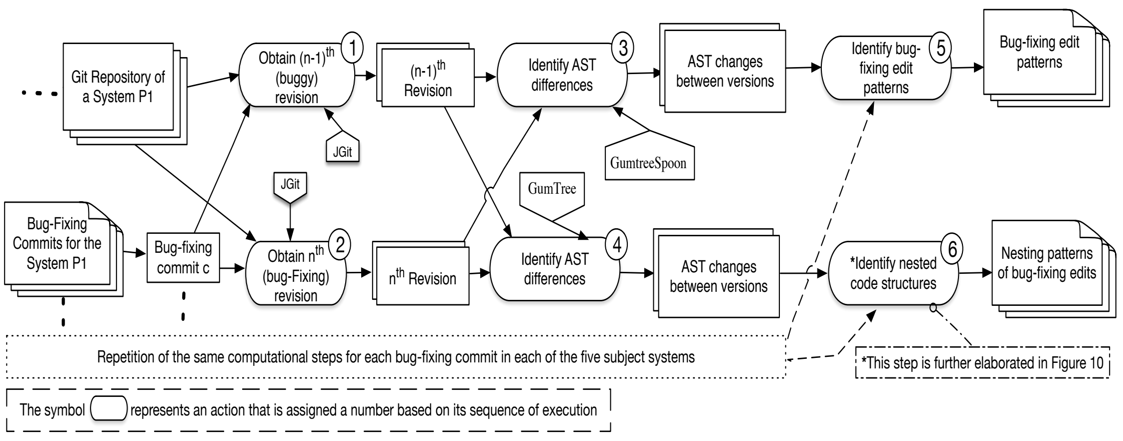

Git Repository of a System P1
Obtain (n-1)th (buggy) revision
1
JGit
(n-1)th Revision
Identify AST differences
3
AST changes between versions
GumtreeSpoon
Identify bug-fixing edit patterns
5
Bug-fixing edit patterns
Bug-Fixing Commits for the System P1
Bug-fixing commit c
JGit
Obtain nth (bug-Fixing) revision
2
nth Revision
GumTree
Identify AST differences
4
AST changes between versions
*Identify nested code structures
6
Nesting patterns of bug-fixing edits
Repetition of the same computational steps for each bug-fixing commit in each of the five subject systems
*This step is further elaborated in Figure 10
The symbol represents an action that is assigned a number based on its sequence of execution

Figure 14.1: Procedural steps to identify *edit* patterns and *nesting* patterns of bug-fixing changes

use of Abstract Syntax Tree (AST) based code differencing tools (e.g., `ChangeDistiller` [322], `Diff/TS` [323] and `GumTree` [319]) for automatic discovery of code-changes and differencing program entities. Previous work on discovering bug-fix patterns remained focused on bug-fixing *edit* patterns, which include bug-fixing changes to source code at a very fine-grained level without capturing those changes' surrounding code contexts such as *nested code structures*. The nested code structure, which is a hierarchy of AST nodes, indicates the location of a bug-fix change in an AST nodes' hierarchy. Nested code structures provide an important code context/aspect of bug-fix changes but remained absent in the studies [319, 324, 321, 325, 326, 318, 327, 328] that identified bug-fixing edit patterns.

In this work, we capture both bug-fixing *edit* patterns and *nesting* patterns (i.e., frequent nested code structures) of bug-fixing edits through an in-depth (quantitative and qualitative) analysis of 4,653 buggy revisions of five software systems drawn from diverse application domains. We organize this chapter around two research questions as follows:

RQ1: *What are the common patterns of bug-fixing edits*? – Here, we explore bug-fixing editings/changes made in source code and identify the bug-fixing edit patterns. We will verify what portion of the identified bug-fixing *edit* patterns are new, and how many of them were previously reported in earlier studies [326, 318, 327].

RQ2: *What are the prominent nested code structures that frequently host bug-fixing edits*? – Here we investigate the frequent *nested code structures* (i.e., nesting patterns) where the bug-fixing edits are located. These *nesting* patterns will complement the *edit* patterns in our understanding of bug-fixing patterns with information about the locations and contexts of individual edits within surrounded nested code structures. Moreover, such AST based nested code structure contexts provide a potential scope to use those along with other code contexts such as *textual similarity of code* [329] to develop an effective technique to automatically locate program faults and repair those.

Contributions: Towards a deeper understanding of bug-fix patterns, this work makes two major contributions:

- We identify a total of 38 bug-fix patterns organized in 14 categories. This is the highest number of bug-fix patterns identified in a single study. Four of these patterns are completely new, and 34 of them confirm those reported in earlier studies.
- We study locations of bug-fix changes in nested code structures and identify 37 *nesting* patterns that hold the majority of the bug-fix edits. These *nesting* patterns are new (i.e., never targeted before), and add a new dimension in our understanding of bug-fix patterns.

14.2 Methodology

The procedural steps of our empirical study are summarized in Figure 14.1. For each subject system, we collect the bug-fixing revisions. Then, for each bug-fixing revision, using AST based code differencing tools, we detect differences between the bug-fixing revision and its immediate previous revision. Collections of such AST differences are then analyzed to detect bug-fixing edit patterns and dominant nested code structures of code changes to fix bugs. In the following, we describe the subject systems and elaborate procedural steps with necessary discussions.

14.2.1 Subject Systems

We study 4,653 revisions of five open-source software systems written in Java. These subject systems, as listed in Table 14.1, are available at GitHub. In Table 14.1, we present the total number of revisions and the number of source lines of code (KLOC) in the last revision. We choose these five subject systems as these systems have variations in application domains, sizes, number of revisions, and are also used in other studies [97, 232].

Moreover, we consciously choose the five subject systems that can be classified in two sets: (i) the first three subject systems, which were never been used earlier to detect bug-fixing *edit* patterns, belongs to the first set and (ii) the second set consists of the remaining two subject systems, which were earlier used in other studies [329]. Such a combination of choosing subject systems provides the opportunity not only to verify whether earlier detected bug-fixing edit patterns exist in our study, but also to identify new bug-fixing edit patterns if they exist.

14.2.2 Collecting Bug-fixing Commits

For the first three systems, we collect the bug-fixing commits identified by Ray et al. [232]. These bug-fixing commits are distinguished through matching keywords (e.g., bug, defect, issue) in the commit messages and are reported to be 96% accurate [232]. To identify the bug-fixing commits in

Table 14.1: Subject Systems

Subject System	Application Domain	KLOC (last rev.)	Total # of Revisions	# of Bug-Fixing Rev.
Netty	Network	1,078	8,534	1,103
Presto	SQL	2,869	11,909	841
Facebook-android-SDK	Social Networking	172	671	133
Accumulo	Distributed Key-value store	458	9,734	1,941
Common-maths	Math library	187	6,971	635
Total over all the systems		4,764	37,819	4,653

the remaining last two systems, we use the same keywords used by Ray et al. [232]. The number of bug-fixing revisions for each system is listed in the last column of Table 14.1.

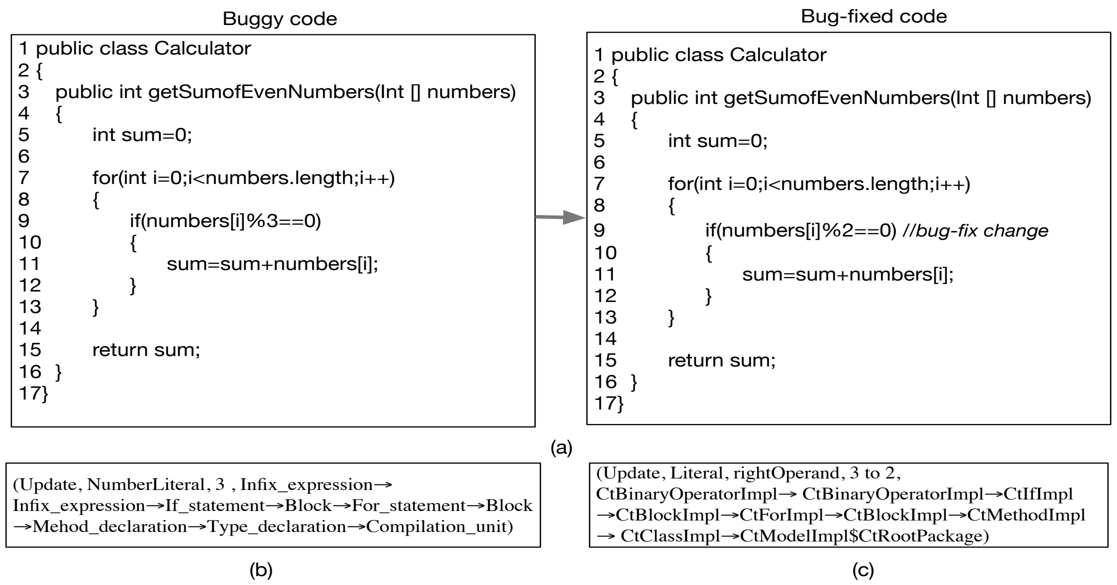
14.2.3 Generating Abstract Syntax Tree of Bug-fixing Changes

Consider a bug-fixing commit C resulting in the n^{th} revision of a system/project $\mathcal{P}_1$. If a particular line of code $\mathcal{L}$ is modified in the bug-fixing commit C, then it implies that the modification is necessary to fix the bug. Thus, the line of code $\mathcal{L}$ in the $(n-1)^{\text{th}}$ revision is considered a buggy line. In other words, we consider the changes between the n^{th} and $(n-1)^{\text{th}}$ revisions of $\mathcal{P}_1$ are buggy. Several other studies [301, 308, 303, 25] also adopted the similar approach for distinguishing buggy source code.

At this level, as shown in Figure 14.1, we obtain the n^{th} and $(n-1)^{\text{th}}$ revisions using `JGit` [307]. Then, we capture bug-fixing changes at the AST [319] level between those two revisions using `GumtreeSpoon` and `Gumtree` separately (see action 03 and 04 in Figure 14.1). Captured AST differences using `GumtreeSpoon` and `Gumtree` are further processed to determine bug-fixing edit patterns and nesting patterns, respectively (see action 05 and 06 in Figure 14.1).

Before describing how we identify bug-fixing edit patterns (in Section 14.3) and nesting patterns (in Section 14.4), in the following, we discuss and compare the outputs of `GumTree` and `GumtreeSpoon` to develop background/context that helps in understanding the rest of the content of the chapter.

Understanding of `GumTree`'s output. For each action/change in a node, `GumTree` generates four major attributes: (i) *action name* (e.g., ins, del, upd or, mov) (ii) *label-* that indicates text/name of the changed node (iii) type of the changed node (e.g., changed node can be a simple `variable name` or an `expression`) and (iv) *nested code structure (NCS)-* the tree/hierarchy of parent nodes



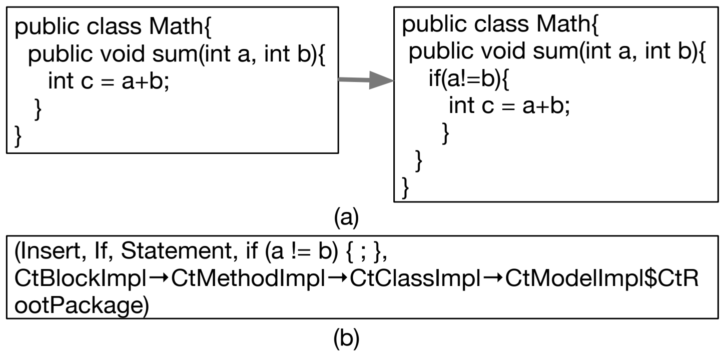
```
1 public class Calculator
2 {
3   public int getSumofEvenNumbers(Int [] numbers)
4   {
5        int sum=0;
6
7        for(int i=0;i<numbers.length;i++)
8        {
9             if(numbers[i]%3==0)
10            {
11                 sum=sum+numbers[i];
12            }
13       }
14
15       return sum;
16  }
17}
```

```
1 public class Calculator
2 {
3   public int getSumofEvenNumbers(Int [] numbers)
4   {
5        int sum=0;
6
7        for(int i=0;i<numbers.length;i++)
8        {
9             if(numbers[i]%2==0) //bug-fix change
10            {
11                 sum=sum+numbers[i];
12            }
13       }
14
15       return sum;
16  }
17}
```

(Update, NumberLiteral, 3 , Infix_expression→Infix_expression→If_statement→Block→For_statement→Block→Mehod_declaration→Type_declaration→Compilation_unit)

(Update, Literal, rightOperand, 3 to 2, CtBinaryOperatorImpl→ CtBinaryOperatorImpl→CtIfImpl→CtBlockImpl→CtForImpl→CtBlockImpl→CtMethodImpl→ CtClassImpl→CtModelImpl$CtRootPackage)

Figure 14.2: (a) Changing a `literal` in an `if` statement to fix a bug and the presentations of the bug-fixing change using (b) `GumTree` and (c) `GumtreeSpoon`



```
public class Math{
  public void sum(int a, int b){
    int c = a+b;
  }
}
```

```
public class Math{
 public void sum(int a, int b){
   if(a!=b){
     int c = a+b;
   }
 }
}
```

(Insert, If, Statement, if (a != b) { ; }, CtBlockImpl→CtMethodImpl→CtClassImpl→CtModelImpl$CtRootPackage)

Figure 14.3: (a) Adding an `if` statement as a precondition to fix a bug and (b) corresponding representation of the bug-fixing change using `GumtreeSpoon`

of the changed node, which indicates the location of the changed node in an AST. We use these four attributes: *(action name, node type, label, NCS)* to represent a changed AST node.

Let's assume, there is a bug in a piece of code presented at the left side of the arrow sign in Figure 14.2(a). The bug resides in line number nine where a developer uses `literal` '3' instead of `literal` '2'. The buggy code is fixed in the bug-fix revision, which is presented at the right side of the arrow sign in Figure 14.2(a). If those two revisions are given to `GumTree`, it will generate differences between the provided revisions, which can be presented using a tuple of four attributes as shown in Figure 14.2(b).

From Figure 14.2(b), it is easily understood that a `NumberLiteral` is *updated* to fix the bug. From the NCS (the last attribute) of the updated node, we see the `NumberLiteral` is a part of two `infix_expressions` (i.e., == and %), which reside in an `if` statement. Again, the `if` statement resides in a block under a `for` statement. The `for` statement is a part of a block inside a *method*. The method resides inside a *type declaration* (i.e., a class) and *compilation unit* is always the root of an NCS. Here, it is noticeable that an NCS represents a *sequence*, where the root and all internal nodes have only one child except the leaf node, which has no child.

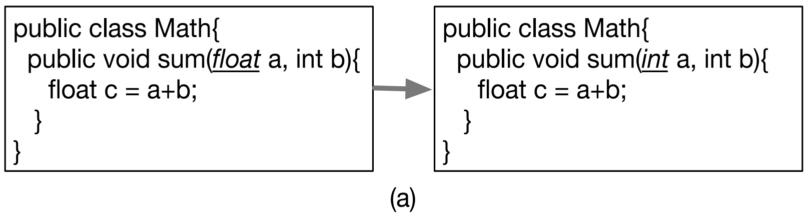
Understanding of `GumtreeSpoon`'s output. While the outputs of `GumtreeSpoon` are almost similar to the outputs of `GumTree`, there are some fundamental differences exist between their outputs. First, `GumtreeSpoon` provides a changed node's role in its immediate parent or node (i.e., *role in parent*), which helps in understanding code changes' patterns. For example, `GumtreeSpoon` indicates that the changed `literal`'s role in its parent is *rightOperand*. How the attribute *role in parent* helps in determining bug-fixing edit patterns is elaborately described in Algorithm 1 presented latter in this chapter.

Second, `GumtreeSpoon` provides *modified source code* as opposed to the *label* provided by `GumTree`. We find modified source code is more helpful to understand bug-fixing edit patterns (see Section 14.3.2) instead of a label. Thus, we use a tuple of five attributes such as (*action name, node type, role in parent, modified source code, nested code structure*) to represent a code change using `GumtreeSpoon`'s output as shown in Figure 14.2(c). It is noticeable in Figure 14.2(b) and 14.2(c) that naming conventions of the nodes are different between `GumTree` and `GumtreeSpoon`.

Finally, while `GumTree` provides fine-grained level differences, `GumtreeSpoon` generates summary/concise level outputs of code changes that help in understanding bug-fixing edit patterns conveniently. For example, the code changes shown in Figure 14.3(a), is represented using `GumtreeSpoon`'s output in Figure 14.3(b). From Figure 14.3(b), we see that only node `if` is inserted, thus `GumtreeSpoon` ignores other fine-grained level changes such as additions of `conditional operator` and variables (e.g., `a` and `b`). In contrast to that, `GumTree's` outputs indicate that five nodes are inserted: *insert block*, *insert ifStatement*, *insert infixExpression* (i.e., ==), and *insert simpleNames* (i.e., variables `a` and `b`). While these detailed, in-depth, and verbose outputs provided by `GumTree` are suitable to analyze nesting patterns in deeper levels to answer RQ2, the concised outputs of `GumTreeSpoon` are required for analyzing bug-fixing edit patterns to answer RQ1.

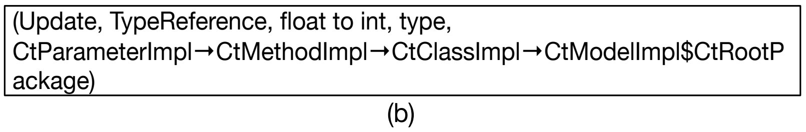
14.3 Bug-fixing Edit Patterns

Once we have the `GumtreeSpoon`'s outputs for the bug-fixing changes, we aim to identify the bug-fixing edit patterns defined by Pan et al. [318]. Pan et al. have defined a set of 27 bug-fixing edit patterns divided in nine categories: If-related (IF), Method Calls (MC), Sequence (SQ), Loop (LP), Assignment (AS), Switch (SW), Try (TY), Method Declaration (MD) and Class Field (CF). Their study has identified the highest number of bug-fixing edit patterns in a single study. Moreover, according to the number of citations, this is one of the most important papers on bug-fix edit patterns, thus it becomes a benchmark for the studies related to bug-fixing edit patterns' detection. In the rest



```
public class Math{
  public void sum(float a, int b){
    float c = a+b;
  }
}
```

```
public class Math{
  public void sum(int a, int b){
    float c = a+b;
  }
}
```



```
(Update, TypeReference, float to int, type,
CtParameterImpl→CtMethodImpl→CtClassImpl→CtModelImpl$CtRootP
ackage)
```

Figure 14.4: (a) Updating parameter type (`float` to `int`) of a method to fix a bug and (b) corresponding representation of the bug-fixing change using `GumtreeSpoon`

of this chapter, we use the term *PanPattern* to refer to a pattern identified by Pan et al. [318]. We also verify whether `GumtreeSpoon` is able to identify any new bug-fixing edit patterns as opposed to the PanPatterns in our dataset.

14.3.1 Making Sense of `GumtreeSpoon`'s Outputs

Here we manipulate the `GumtreeSpoon`'s outputs using Algorithm 1 to make those more obvious for our analysis. Based on a preliminary investigation, we find that a code change belongs to or impacts the node that is an immediate previous node of the first occurrence of a `block` node in an NCS. For example, from the bug-fixing change presented in Figure 14.2(a), it is not obvious that the change occurs in an `if` statement until we see the immediate previous node of the first `block` node (which is indeed an `if` node) in the NCS given in Figure 14.2(c) generated by `GumtreeSpoon`.

When an NCS contains at least one `block` node, we determine the pattern of a bug-fixing change using the procedure described in Algorithm 1, Lines 2–8. An NCS starts with a `block` node if an insertion or deletion or update is performed on a node, which is not contained in or associated or linked with any other node within its block. As shown in Figure 14.3(b), a node `If` is inserted and the NCS starts with a `block` node. The pattern for this type of bug-fix changes is determined using the action (i.e., ins/del/upd) performed on a node to change code, and the name of the changed node as shown in Algorithm 1, Lines 3–4.

Another category of bug-fix changes contains those type of patterns where the implementation of a node is updated by performing an action on any other nodes, which are contained in or associated or linked with the implementing node within its block. As shown in Figure 14.2(a), a `literal` node, which is contained in an implementing `if` node, is updated where both the nodes (i.e., `if` and `literal`) reside in the same block. In this case, the bug-fixing edit pattern is determined using action, changed node name, and the immediate previous node's name of the first occurrence of a `block` node in an NCS (see Algorithm 1, Lines 6–7).

The third category of bug-fix edit patterns does not have any `block` node in the NCSes for changes in the definitions of class or interface members such as addition/removal of class fields or methods or changes in the types of parameters of methods. As shown in Figure 14.4(a), a developer updates type of a parameter from `float` to `int` to fix a bug. Figure 14.4(b) represents the change using the output of `GumtreeSpoon` where the NCS does not have any `block` node. For this case, we identify a bug-fixing edit pattern by incorporating a changed node's *role in parent* attribute, and consider the first node in the NCS as the location of the change.

If a class member (e.g., method, variable) or a parameter of a method is changed, we use action, node name, and the first node in the NCS to determine the pattern of the bug-fix change (see Algorithm 1, Lines 10–14). If the type a class variable or method's parameter is changed, then we determine the location of the change (e.g., type of a class variable or method's parameter) (see Algorithm 1, Line 16), and use that along with action and node name to determine the bug-fix pattern (see Algorithm 1, Line 17).

For the bug-fixing changes presented in Figure 14.2(a), Figure 14.3(a), and Figure 14.4(a), Algorithm 1 will output the patterns *update literal of CtIfImpl*, *insert `if`*, and *update type of a parameter of a method*, respectively. The set of patterns that we identify using Algorithm 1 are termed as *GSPatterns* in the rest of this chapter.

Algorithm 1 Detection of GSPatterns

```
Input: T : a tuple of five attributes generated by GumtreeSpoon for a code change
1  String pattern;

2  if T.NCS.contains("Block") then
3      if T.NCS.startsWith("Block") then
4          pattern←T.action+" "+ T.nodeName;
5      else
6          String IPN←getPreviousNodeOfFirstBlock(T.NCS)  pattern←T.action +" " + T.nodeName + " of "+ IPN
7      end
8  else
9      String FNN←getFirstNodeInNCS(T.NCS)  if T.roleInParent.equals("typeMember") then
10         pattern←T.action +" "+ T.nodeName+" in " + FNN
11     else if T.roleInParent.equals("parameter") then
12         pattern←T.action +" "+ T.nodeName+" in " + FNN
13     else if T.roleInParent.equals("type") then
14         String CFL←getChangeLocation(T.NCS);
15         pattern←T.action +" "+ T.nodeName +" in " + CFL;
16 end
17 return pattern;
```

14.3.2 Mapping GSPatterns to PanPatterns

In most of the cases, a GSPattern can be mapped directly to its corresponding PanPattern. For example, the GSPattern *update literal of CtIfImp* indicates its corresponding PanPattern *change of `if` condition expression* (IF-CC) [318].

However, we have to leverage the attribute *modified source code* to identify PanPatterns from their corresponding GSPatterns in two cases that include: (i) addition of a precondition (i.e., `if`

Table 14.2: Distributions of *PanPatterns* identified in our dataset using `GumtreeSpoon`

Category (Cat)	Pattern (Pat)	# of Pat	% of Pat	Total per Cat	% per Cat
Method Declaration (MD)	Change of method declaration (MD-CHG)	4,116	17.54%	7,687	33.00%
	Addition of a method declaration (MD-ADD)	1,916	8.17%		
	Removal of a method declaration (MD-RMV)	1,655	7.05%		
If-related (IF)	Addition of post-condition check (IF-APTC)	1,975	8.42%	4,877	20.78%
	Removal of an `if` predicate (IF-RMV)	1,588	6.77%		
	Change of `if` condition expression (IF-CC)	761	3.24%		
	Addition of precondition check (IF-APC)	309	1.32%		
	Addition of precondition check with jump (IF-APCJ)	37	0.16%		
	Removal of an `else` branch (IF-RBR)	71	0.30%		
	Addition of an `else` branch (IF-ABR)	136	0.58%		
Method Call (MC)	Method call with different number of parameters or different types of parameters (MC-DNP)	2,402	10.24%	4,639	20.00%
	Change of method call to a class instance (MC-DM)	1,582	6.74%		
	Method call with different actual parameter values (MC-DAP)	655	2.79%		
Class Field (CF)	Addition of a class field (CF-ADD)	1,355	5.77%	3,735	16.00%
	Change of class field declaration (CF-CHG)	1,719	7.33%		
	Removal of a class field (CF-RMV)	661	2.82%		
Assignment (AS)	Change of `assignment` block expression (AS-CE)	1,401	0.97%	1,401	5.97%
Loop (LP)	Change of `loop` predicate (LP-CC)	549	2.34%	549	2.34%
Try (TY)	Addition/removal of `try` statement (TY-ARTC)	491	2.09%	526	2.24%
	Addition/removal of a `catch` block (TY-ARCB)	35	0.15%		
Switch (SW)	Addition/removal of `switch` block branch (SW-ARSB)	50	0.21%	50	0.21%
			Overall total	**23,464**	**100%**

node) check with/without jump statement (e.g., `return`, and `break`) and (ii) changes in a method call. An inserted `if` statement acts as a precondition if it wraps up existing code, otherwise, that will be considered as a new insertion of an `if` node. For any inserted `if` node if we find modified source code contains any lone semicolon (;) in a line, then the inserted `if` statement/node is considered as a precondition. For example, the modified source code, presented in Figure 14.3(b), contains a lone semicolon in the bug-fixing change presented in Figure 14.3(a). The number of such lone semicolons indicates the number of lines wrapped up by a precondition. In addition to semicolon, we also check whether modified source code contains any jump statement such as `return`, `continue`, or `break` to identify if any precondition is added with a jump that corresponds to another PanPattern *addition of a precondition check with a jump* (IF-APCJ). We use the same logic to identify if a piece of code is wrapped up by statements such as `try-catch`, `loop`, or `switch-case`. We hypothesize an inserted `if` is added as postcondition if that is not a precondition.

In the second case, we parse modified source code to extract the method call statements in a buggy revision and its non-buggy revision. Then, for each method call, we extract the method name, arguments, and class name of a method call if available. Then, we compare that extracted information between the buggy and non-buggy method call statements to identify the location where a change occurs to map the change to its corresponding PanPattern. Multiple changes may occur in a method call (e.g., *the return type* can be changed and *an argument can be inserted*) to fix a buggy method call. In such cases, we record all types of changes and use those to identify bug-fixing edit patterns.

Table 14.3: Distributions of `New` *Edit* Patterns Identified in our Dataset using `GumtreeSpoon`

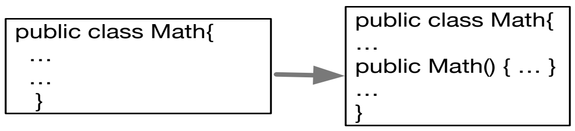
Category (Cat)	**Pattern (Pat)**	**# of Pat**	**% of Pat**	**Total per Cat**	**% per Cat**
Local Variable (LV)	Update implementation of `local variable` (LV-IMPL)	4,043	15.41%	7,709	29.38%
	Addition or deletion of `local variable` (LV-AD)	3,666	13.97%		
Method Call (MC)	Class/target change of method call (MC-TC)	2,881	10.98%	7,531	28.70%
	Addition of new method call (MC-A)	2,754	10.50%		
	Deletion of new method call (MC-D)	1,896	7.23%		
Return (RT)	Update implementation of `return` statement (RT-IMPL)	3,361	12.81%	4,200	16.01%
	Addition or deletion of `return` statement (RT-AD)	839	3.20%		
Assignment (AS)	Addition or deletion of `assignment block` statement (AS-AD)	3,390	12.92%	3,390	12.92%
Constructor (CT)	*Addition or deletion of `constructor` (CT-AD)	578	2.20%	1,013	3.86%
	*Parameter update in `constructor` (CT-Param)	435	1.66%		
Throw (TW)	Update of implementation of `throw` statement (TW-IMPL)	651	2.42%	861	3.28%
	Addition or deletion of `throw` statement (TW-AD)	210	0.80%		
Class or Interface (CI)	Addition or deletion of class or interface (CI-AD)	480	1.83%	480	1.83%
Wrap/Unwrap Code (WU-Code)	Wrap/unwrap code with/from high-level Node (WU-Code)	410	1.54%	410	1.54%
Loop (LP)	Addition and/or deletion of `loop` statement (LP-AD)	405	1.54%	405	1.54%
Catch (CA)	*Addition or deletion of `catch` variable (CA-AD)	130	0.50%	130	0.50%
Enum (EN)	*Addition or deletion of `enum` statement (EN-AD)	112	0.43%	112	0.43%
***Completely new bug-fixing edit patterns identified in this study**			**Overall total**	**26,241**	**100%**

14.3.3 Dominant Bug-fixing Edit Patterns

14.3.3.1 Detected PanPatterns

By processing `GumtreeSpoon`'s outputs we are able to detect 21 types of PanPatterns distributed in seven categories presented in Table 14.2. The abbreviations/initials of the categories and patterns' names are given in the same table. The MD category contains the highest number of bug-fixing changes (33.00%), followed by the IF (20.78%), MC (20.00%), and CF (16.00%) categories. Noticeable, the first four categories consist of almost 90% of bug-fixing changes. Category SW experiences the lowest number of bug-fixing changes (0.21%) preceded by LP and TY categories that consist of only 2.34% and 2.24% of the total number of PanPatterns, respectively.

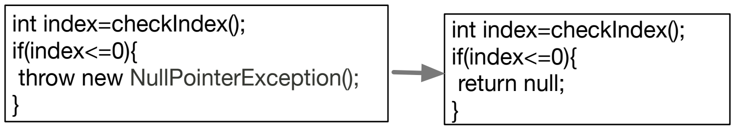
The pattern MD-CHG experiences the highest number of bug-fixing changes (17.54%) followed by the patterns MC-DNP (10.24%) and IF-APTC (8.42%). Interestingly, those three patterns are from three distinct categories. MD-ADD, CF-CHG, and MD-RMV are the next three patterns that experience the highest number of bug-fixing changes (range from 7.05% to 8.17%) after those formerly mentioned three patterns. The patterns MC-DM and IF-RMV experience almost equal amount of bug-fixing changes (≈06.70%). Surprisingly, the patterns IF-APCJ, IF-RBR and IF-ABR from IF-related category together contribute only 1.04% of the total PanPatterns. Except the patterns SW-ARSB and TY-ARCB (that contribute only 0.21% and 0.15% of the total number of PanPatterns, respectively), the proportions of the remaining PanPatterns range from 1.32% to 5.77%.



```
public class Math{
  ...
  ...
  }
```

```
public class Math{
...
public Math() { ... }
...
}
```

Figure 14.5: Insertion of a `constructor` to fix a bug



```
int index=checkIndex();
if(index<=0){
 throw new NullPointerException();
}
```

```
int index=checkIndex();
if(index<=0){
 return null;
}
```

Figure 14.6: Deletion of a `throw` statement to fix a bug

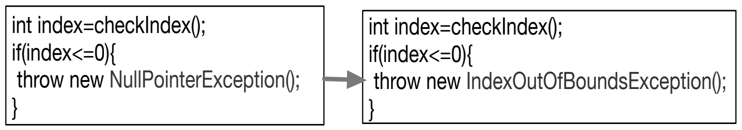
14.3.3.2 New bug-fixing edit patterns

Using `GumtreeSpoon` we identify 17 types of new bug-fixing edit patterns in 11 categories presented in Table 14.3. Here, we indicate those bug-fixing edit patterns as new, which are not defined in PanPatterns. Although some of those 17 bug-fixing edit patterns are already identified in different studies [325, 326, 327], we identify completely four new bug-fixing edit patterns as indicated in Table 14.3. In the following, we briefly define the new patterns, which are relatively complex, while some of those patterns can be easily understood from their names such as addition or deletion of a method call (MC-AD) or a class/interface (CI-AD).

Addition or deletion of node N_1 (N_1-AD). This type of pattern consists of addition or deletion of a node N_1 where N_1 ∈ {`constructor`, `throw`, `loop`, `enum`, `return`, `local variable`, `assignment`}. For example, in Figure 14.5, we see a `constructor` is inserted in a class to fix a bug. Again, in Figure 14.6, a `throw` statement is deleted to fix another bug.

For each of the seven nodes we define seven patterns such as (i) addition or deletion of `constructor` (CT-AD), (ii) addition or deletion of `throw` (TW-AD), (iii) addition or deletion of `loop` (LP-AD), (iv) addition or deletion of `enum` (EN-AD), (v) addition or deletion of `return` (RT-AD), (vi) addition or deletion of `local variable` (LV-AD) and (vii) addition or deletion of `assignment` (AS-AD).

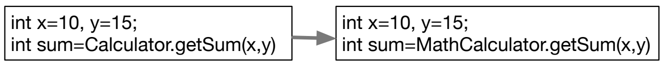
Update implementation of node N_2 (N_2-$IMPL$). In this type of pattern, an implementation of a node N_2 is updated by performing actions on other nodes associated with the implementing node. For example, in Figure 14.7, we see the implementation of a node `throw` is changed by updating an associated node `NullPointerException()` to `IndexOutOfBoundsException()` to fix a bug. Here N_2 ∈ {`throw`, `return`, `local variable`}. Again, for each of the three nodes, we define



```
int index=checkIndex();
if(index<=0){
 throw new NullPointerException();
}
```

```
int index=checkIndex();
if(index<=0){
 throw new IndexOutOfBoundsException();
}
```

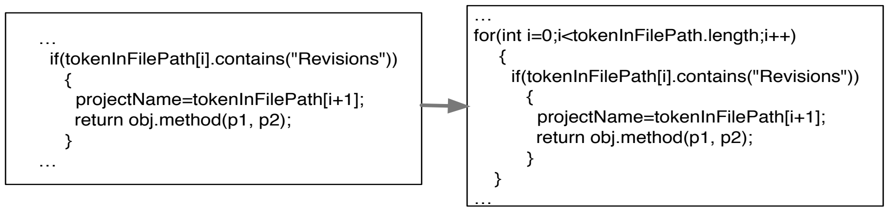
Figure 14.7: Updating implementation of a `throw` statement to fix a bug



```
int x=10, y=15;
int sum=Calculator.getSum(x,y)
```

```
int x=10, y=15;
int sum=MathCalculator.getSum(x,y)
```

Figure 14.8: Changing class/target of a method call to fix a bug



```
...
 if(tokenInFilePath[i].contains("Revisions"))
   {
    projectName=tokenInFilePath[i+1];
    return obj.method(p1, p2);
   }
...
```

```
...
for(int i=0;i<tokenInFilePath.length;i++)
   {
    if(tokenInFilePath[i].contains("Revisions"))
      {
       projectName=tokenInFilePath[i+1];
       return obj.method(p1, p2);
      }
   }
...
```

Figure 14.9: Wrapping up existing code using a `for` loop to fix a bug

three patterns such as (i) update implementation of `throw` (TW-IMPL), (ii) update implementation of `return` (RT-IMPL) and (iii) update implementation of a `local variable` (LV-IMPL).

Class/target change of method call (MC-TC). This pattern contains those types of bug-fixing changes where the class or target of a method call is changed to fix a bug. As shown in Figure 14.8 where the class of the method `getSum` is changed to fix a bug.

Parameter update in Constructor (CT-Param). Similar to pattern MD-CHG, parameters of a `constructor` can be changed and such changes belong to this pattern.

Wrap/unwrap code with/from high-level Node (WU-Code). This pattern of code changes consists of wrapping or unwrapping existing code with/from high-level nodes. The set of high-level nodes h includes {`if`, `for`, `foreach`, `while`, `do-while`, `synchronized`, `try-catch`} that can contain other types of nodes. As shown in Figure 14.9 a piece of existing code is wrapped up inside a `for` loop to fix a bug.

14.3.3.3 Comparative frequencies of the new patterns

As shown in Table 14.3, the category LV consists of the highest number of bug-fixing changes (29.38%) followed by the categories MC (28.70%), RT (16.01%), and AS (12.92%). Again, these four categories consist of almost 90% of newly identified bug-fixing changes.

The pattern LV-IMPL experiences the highest number of bug-fixing changes (15.41%) followed by the pattern LV-AD (13.97%). The patterns AS-AD and RT-IMPL experience almost equal amount of bug-fixing changes ($\approx$13%). Next three patterns MC-TC, MC-A, and MC-D are from MC categories experience 10.98%, 10.50%, and 7.23% bug-fixing changes, respectively. Those seven patterns together contribute almost 84% of bug-fixing changes. The patterns EN-AD, CA-AD, and TW-AD represent the three lowest bug-fixing changes (below 1.00%). The amounts of the rest of the patterns range from 1.50% to 3.20%.

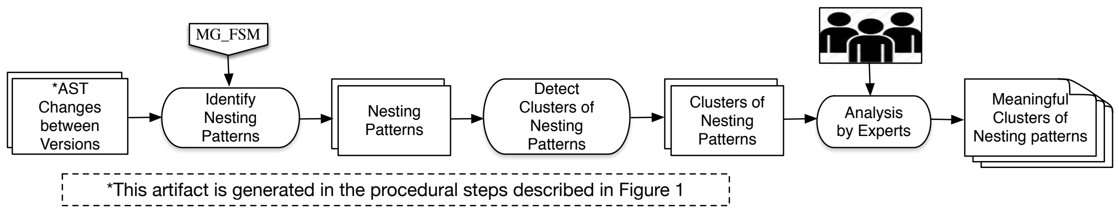

MG_FSM
*AST Changes between Versions
Identify Nesting Patterns
Nesting Patterns
Detect Clusters of Nesting Patterns
Clusters of Nesting Patterns
Analysis by Experts
Meaningful Clusters of Nesting patterns
*This artifact is generated in the procedural steps described in Figure 1

Figure 14.10: Steps to identify nesting patterns that host bug-fixing edits

14.4 Dominant Nesting Patterns

In Figure 14.10, we depict the steps required to detect nesting patterns by capturing the NCSes that frequently host the bug-fixing edits. The steps are briefly described in the following subsections.

14.4.1 Sequential Pattern Mining of Nested Code Structures

In Section 14.2.3, we see that an NCS or parents' tree structure hosting a bug-fixing edit can be presented as a *sequence* of parents nodes. Thus, to identify nesting patterns (i.e., dominant NCSes), we use a sequential pattern mining technique. Sequential pattern mining identifies a set of subsequences or patterns that occur in some percentage or, with minimum support of the input sequences. Any patterns that are found to have support value above or equal to the value of minimum support are said to be dominant patterns. Here, using a sequential pattern mining algorithm, we identify the nesting patterns that are dominant.

However, since a frequent long sequence contains a combinatorial number of frequent subsequences, such mining will generate an exhaustive set of patterns, which will be highly expensive in terms of time and space. To reduce the number of smaller sub-patterns that are found by the sequential pattern mining algorithm, we require that a mining algorithm produces *closed* or *maximal patterns* [330, 331], where sub-patterns that are contained within longer patterns are ignored. As we aim to identify nesting patterns of bug-fixing changes, we find the maximal pattern mining is preferable in our case.

While there are few commonly used sequential pattern mining algorithms available [332], we use recently proposed `MG-FSM` algorithm [333] that meets our requirement to specify constraints such as pattern type (e.g., closed or maximal) and gap constraint between two successive nodes. Moreover, the algorithm is capable in parallel running using map-reduce (Hadoop) functionality. We run the tool by allowing no gap between two successive nodes to determine maximal patterns that have at least 1,000 occurrences. The tool delivers total of 534 sequences that are dominant. We exclude those patterns that do not have at least one block node to make sure containment of a node inside another node. Finally, we have 385 nesting patterns that we use for clustering as follows.

14.4.2 Clustering of Nesting Patterns

At this step, we cluster similar types of nesting patterns in groups. Such clustering provides a convenient way to examine the identified nesting patterns where developers commonly perform bug-fixing changes.

14.4.2.1 Selection of clustering algorithm

To cluster nesting patterns, we use k-medoids [334] algorithm that is a variant of k-means [335] algorithm. While both the k-means and k-medoids algorithms break a dataset up into groups, the latter algorithm uses existing points in the dataset as cluster centroids. Moreover, k-medoids is known for more robustness against noises and outliers compared to k-means [334]. In addition, in our dataset k-means cannot be used directly because numerical operations, such as addition and division, cannot be performed on two patterns, which consist of strings [333].

14.4.2.2 Determining optimal number of clusters

To determine the optimal number of clusters (i.e., k in k-medoids), we choose to use *gap statistic* [336] method over other available options such as *elbow* and *silhouette* methods. The reason why we choose this method as it can be applied to any clustering method (i.e., k-medoids, k-means clustering, and hierarchical clustering). Using the gap statistic, we find the number of optimal clusters is 10 for our data.

14.4.2.3 Defining a distance function for k-medoids

We use the *Longest Common Subsequence* (LCS) based string metric to measure the distance between a pair of mined nesting patterns. We define the distance function for any two mined nesting patterns S_1 and S_2 as follows.

$$D_{LCS}(S_1, S_2) = 1 - \frac{\mid LCS(S_1, S_2) \mid}{max(\mid S_1 \mid, \mid S_2 \mid)}$$

Here, S_1 and S_2 are two finite sequences of nodes, $\mid LCS(S_1, S_2) \mid$ is the length of the longest common subsequence(s) of S_1 and S_2 and $max(\mid S_1 \mid, \mid S_2 \mid)$ is the length of the longest sequence of S_1 and S_2.

We measure LCS metric by using the Python package `python-string-similarity` [337] that implements the said metric. Then, to cluster nesting patterns, we run the open source implementation of k-medoids algorithm provided in Python clustering package `Pycluster 1.49` [338].

At this point, we have 10 clusters of nesting patterns. As the mechanism to generate cluster is based on the names of the AST nodes (i.e., text based clustering), the clusters are required to be interpreted/characterized by human experts to gain meaningful insights of the structures of the nesting patterns in the clusters.

14.4.3 Characterization of the Clusters by Experts

The first two authors are presented with a listing of all the patterns in each cluster, and asked to characterize those patterns in terms of their nodes' hierarchies. By observing nodes' hierarchies of the patterns, they create two sets of nodes: (i) a set of low level nodes l, where $l\epsilon$ {`return`, `expression`, `throw`, `variable declaration`, `assignment`} and (ii) another set of high level nodes h defined in Section 14.3.3.2.

Each author aims to identify if a low label node's block from l is contained in a high-level node block from h. Such an identification is represented as a pattern/cluster l block→h block. For example, if a block of `return` is located inside an `if` block, then the authors label that hierarchy as `return` block→`if` block. If a higher-level node block h_1 is contained in another high-level node block h_2, then that pattern is categorized as 'Compound' and represented as h_1 block→h_2 block. In a similar fashion, deeper-levels' containments/hierarchies can also be presented (see the third column of the last row in Table 14.4).

Cohen's kappa coefficient κ [339] is used to measure agreement between two authors in charatcterizing patterns. κ value 0.79 indicates high-level agreement on the characterization of the patterns of the clusters. For each disagreement, authors discuss between them, and if necessary, they verify raw data to come to an agreement. Such discussions result in an unconventional pattern that has chained method invocations (e.g., m1().m2().m3()) in a single block (see the fourth category in Table 14.4). Finally, total of 37 meaningful clusters is identified in six categories: (i) IF-related (IF), (ii) Try-Catch (TY-CA) (iii) Loop (LP) (iv) Chained Method Invocation (CMI) (v) Synchronize (SYN) and (vi) Compound (COM) as presented in Table 14.4. As per the definition of the category COM, the category SYN falls in the COM category, although the authors decide to create a separate category for it. As all the patterns are ended with `Method_declaration→Type_declaration→Compilation_unit`, we truncate that for better presentation.

14.4.4 Mining Results

There are nine types of nesting patterns or clusters belong to the IF category that represents the largest amount (40.72%) of the total number of patterns followed by the COM category that consists of 15 types of patterns contribute to 31.82% of the total number of patterns. The categories TY-CA and LP contribute 9.89% and 8.02% of total patterns, respectively, followed by the CMI category that consists of 5.93% of total patterns. The number of patterns belongs to the SYN category is the lowest (3.60%).

Table 14.4: Dominant *Nesting* Patterns That Host Bug-Fixing Edits

Category (Cat)	**Cluster/Nesting Pattern (Pat) ID**	**Mined Nesting Patterns**	**# of Pat**	**% of Pat**	**Total per Cat**	**% per Cat**
IF-related (IF)	01	`if` block→`if` block	27,306	10.93%	101,691	40.72%
	02	Method invocation block→`if` block	19,430	7.78%		
	03	`expression` block→`if` block	16,838	6.74%		
	04	`assignment` block→`if` block	10,250	4.10%		
	05	`variable declaration` block→`if` block	9,996	4.00%		
	06	`throw` block→`if` block	9,274	3.71%		
	07	`return` block→`if` block	4,923	1.97%		
	08	Method invocation block as `expression` →`if` block	3,771	1.51%		
	09	Nested `If` (with/without `else`)	2,634	1.05%		
Try-Catch (TY-CA)	10	block→`try` block	12,321	4.93%	24,709	9.89%
	11	`variable declaration` block→`catch` block	5,858	2.34%		
	12	Method invocation block→`try` block	3,197	1.28%		
	13	`throw` block→`try-catch` block	1,248	0.49%		
	14	`expression` block →`try` block	1,048	0.41%		
	15	`try` block→`try` block	1,037	0.41%		
Loop (LP)	16	`variable declaration` block→`loop` block	10,202	4.08%	20,029	8.02%
	17	Method invocation block→`loop` block	6,001	2.40%		
	18	`expression` block→`loop` block	2,316	0.92%		
	19	`assignment` block→`loop` block	1,510	0.60%		
Chained Method Invocations (CMI)	20	Chained method invocations	14,828	5.93%	14,828	5.93%
Synchronize (SYN)	21	`if` block→`synchronized` block	4,888	1.95%	8,986	3.60%
	22	`loop` block→`synchronized` block	4,098	1.64%		
Compound (COM)	23	`if` block→`loop` block	27,583	11.04%	79,484	31.82%
	24	`if` block→`try` block	13,328	5.33%		
	25	`loop` block→`if` block	6,457	2.58%		
	26	`loop` block→`try` block	4,818	1.92%		
	27	`try` block→`if` block	3,725	1.49%		
	28	`expression` block→`loop` block→`if` block	3,687	1.47%		
	29	`loop` block→`loop` block	3,386	1.35%		
	30	`try` block→`loop` block	2,205	0.88%		
	31	`if` block→`loop` block→`if` block	2,141	0.85%		
	32	`switch` block→`if` block	1,413	0.56%		
	33	`if` block→`loop` block→`try` block	1,189	0.47%		
	34	`if` block→`switch` block	1,161	0.46%		
	35	`switch` block→`loop` block	1,145	0.45%		
	36	`variable declaration` block→`if` block→`loop` block	1,118	0.44%		
	37	`expression` block→`loop` block→`try` block	1,026	0.41%		
				Overall Total	**249,727**	100%

By inspecting individual clusters, we find some interesting patterns that can not be identified without considering hierarchies of NCSes. The 23rd cluster (i.e., `if` block→`loop` block) is the most bug prone pattern as it experiences the highest number (27,583) bug-fix changes followed by the first cluster `if` block→`if` block, which is slightly lower than the former cluster. Noticeable, the number of bug-fix changes in a pattern l block→`if` block is always higher than a pattern l block→h' block, where $h' = h - \texttt{if}$. For example, the number of occurrences of the pattern `expression` block→`if` block is higher than the number of occurrences of the pattern `expression` block→`loop` block. Recalling that l represents the set of low level nodes.

It is very interesting that `throw` blocks inside `if` blocks are more bug-prone than `throw` blocks inside `try-catch` blocks. Although in Table 14.2 we see the number of changes in the category Try (TY) is very low, the opposite result is observed in Table 14.4, where category related to Try (TY-CA) experiences the second highest bug-fix changes among the categories of simple high-level nodes. It means that pieces of code inside `try-catch` frequently experience bug-fix changes. A similar observation is also applicable to the SYN category.

Surprisingly, only five clusters among 37 clusters contain three-levels containment (see clusters 28, 31, 33, 36, and 37), which consist of only 3.64% of all patterns. The pattern `expression` block→`loop` block→`if` block consists of almost 50% of all those three-levels patterns. No pattern is found that contains more than three-levels containment.

14.5 Threats to Validity

Construct Validity. For the first three projects, we use the bug-fixing commits that are identified by Ray et al. [232]. To collect those buggy commits, they used a technique similar to the approach of Mockus and Votta [340]. A similar approach is also used for detecting bug-fixing commits of the last two projects. There is a possibility that some portions of the bug-fixing commits may be general commits (e.g., new feature, and improvement), thus, our data may contain some false positives. Having said that, Ray et al. found 96% accuracy of their approach to collect those bug-fixing commits that minimized the threat substantially.

To detect bug-fixing edit patterns, we have considered only nodes found before the occurrence of the first block node (if available) in a NCS. Someone may be skeptical in capabilities of detecting bug-fixing edit patterns using such an approach. However, our approach is found successful in detecting not only existing bug-fixing edit patterns but also new bug-fixing edit patterns from code bases. To detect nesting patterns, we have identified maximal patterns instead of closed patterns of nested code structures. Closed sequential pattern mining algorithms remove all patterns that exist within other identified patterns and occur at the same support level, while maximal pattern mining removes sub-patterns regardless of the support level. For our problem, the maximal pattern mining is preferable, as we have aimed to identify deeper nested code structures instead of sub-structures (i.e., sub-patterns).

To detect maximal patterns, we have allowed no gap between two nodes and only considered those as patterns, which have at least 1,000 occurrences. To detect exact nested code structures, it is obvious that setting “no gap between two nodes" is the best choice. Although the setting of 1,000 as the threshold can be criticized, we have found the setting was capable of retaining over 70% bug-fixing transactions, while minimized human efforts in detecting meaningful clusters.

In detecting patterns, we have excluded any *mov* actions on nodes found in ASTs for bug-fixing changes. Typically, an insertion or a deletion of a node caused moving of other nodes in an AST where moved nodes remain unchanged. Thus, such unchanged nodes (i.e., *mov* actions) can be ignored, although it is still a threat in detecting patterns related to *mov* action.

Internal Validity. The correctness of our analysis depends on both `GumTree` and `GumtreeSpoon` tools, which are used to answer *RQ2* and *RQ1*, respectively. The former tool outperforms the state-of-the-art tool `ChangeDistiller` by maximizing the number of AST node mappings, minimizing the edit script size, and detecting better move actions [319]. Moreover, `ChangeDistiller` works at the statement level, preventing the detection of certain fine-grain patterns.

Similar to `GumtreeSpoon`, there is another tool `CLDIFF` [320] also available. Between `CLDIFF` and `GumtreeSpoon`, we selected the latter tool for addressing *RQ1*, because, from a sample test run, we revealed that `CLDIFF` failed in distinguishing changes to method parameters at its concise level outputs. But, for this work on bug-fixing edit patterns, this capability is very important for capturing subtle changes in a code made for bug-fixing [318].

The library `JGit` [307] is used in our work to extract a buggy and its previous revisions. The library is applied for a similar purpose in other studies [311, 312], which has brought confidence in us to use that.

External Validity. Although our study includes a large number of revisions of five subject systems, all the systems are open-source and written in Java. Thus, the findings from this work may not be generalizable for industrial systems and source code written in languages other than Java.

Reliability. The methodology of this study including the procedures for data collection and analysis are documented in this chapter. The subject systems being open-source, are freely accessible while the tools `GumTree`, `GumtreeSpoon`, `MG-FSM`, and library `JGit` are also available online. Therefore, it should be possible to replicate the study.

14.6 Related Work

14.6.1 Identifying bug-fixing edit patterns

Pan et al. [318] manually identified a set of 27 bug-fixing edit patterns (i.e., PanPatterns) by exploiting textual differences between buggy and non-buggy programs. A similar approach was also applied by Yue et al. [328] to identify 11 bug-fixing edit patterns, however, they used clustering technique to minimize manual efforts. Both studies are subject to few limitations: (i) obviously identifying all possible bug-fixing edit patterns using manual effort is a daunting task, which arises possibility of failure in discovering all types of bug-fixing edit patterns, and (ii) they used textual differences between buggy and non-buggy programs to identify bug-fixing edit patterns, which is reported to be limited in detecting bug-fixing edit patterns [319].

Despite few limitations, the study of Pan et al. is the most influential work (in terms of citation numbers) in the related area and till now they have identified the highest number of bug-fixing edit patterns in code bases. That is why we started our work by targeting this study to identify bug-fixing edit patterns (while at the same time we kept an eye on any new or unseen patterns). Instead of textual difference, we have used a state-of-the-art AST based code differencing tool `GumtreeSpoon` and developed a fully automated approach to detect bug-fixing edit patterns. Moreover, we have detected 37 bug-fixing edit patterns, which is the highest among all the studies [324, 326, 325, 318, 327] carried out to identify bug-fixing edit patterns till date. In addition, we have identified four new patterns in Constructor, Catch and Enum categories (see Table 14.3).

Kim et al. [324] also employed manual efforts to identify a set of 10 dominant bug-fixing edit patterns (known as PAR templates). However, to collect bug-fixing patches they used `Kenyon` framework [341] and clustered those patches using `groums` [342] to minimize human efforts. Again, Sobreira et al. [327] manually analyzed only 395 patches collected from *Defects4J* [343] project and identified 25 bug-fixing edit patterns. Although the latter study identified the second highest number bug-fixing edit patterns after PanPatterns and identified few new patterns too, the work was conducted on a very small dataset. Thus, additional studies are required to verify their newly identified bug-fixing edit patterns related to `return`, `throw` and wrap/unwarp-code using larger datasets and our work can be considered such a work that verifies those new patterns are dominant in bug-fixing changes. Moreover, they identified 33 instances of a pattern where `throw` blocks reside in `if` blocks. The sixth cluster in Table 14.4 confirms such finding of their study. In addition, in a very recent study, Tufano et al. [344] manually identified new patterns of only five instances related to `synchronized` blocks' additions and deletions. The question is "can we consider those as patterns with only five instances?". Our study can answer this question as 'yes' by observing the 21st and 22nd clusters in the SYN category presented in Table 14.4.

To overcome the problems of manual approaches, automatic techniques are developed to identify bug-fixing edit patterns. Martinez and Monperrus [326] identified 20 bug-fixing edit patterns using the AST based code differencing tool `ChangeDistiller` [322]. In our study, we have used the tool `GumtreeSpoon` [345] developed based on `GumTree` [319], which is more accurate than `ChangeDistiller`.

Soto et al. [346] conducted a study of bug-fixing commits in Java projects. Campos et al. [98] characterized the prevalence of the five most common bug-fixing edit patterns related to IF category. However, the studies of Soto et al. and Campos et al. limited the searching of bug-fixing edit patterns within PAR templates and patterns identified by Pan et al., respectively. In our case, we have remained open to identify any bug-fixing edit patterns exist in code bases. Osman et al. [312] applied a semi-automated approach to identify five bug-fixing edit patterns. In contrast to their approach, we have used a fully automated technique to identify 37 bug-fixing edit patterns.

Hanam et al. [347] developed the `BugAID` technique for discovering bug-fixing edit patterns in JavaScript. Sudhakrishnan [348] identified 25 bug-fixing edit patterns in Verilog (a hardware description language). Long and Rinard [349] learned a probabilistic model of correct patch from bug-fixing changes of C code. In contrast to their studies, we have studied five Java projects. While all the previously mentioned studies identified bug-fixing edit patterns, none of the studies conducted any nested code structures analysis to detect nesting patterns, which is a *novel contribution* of our study.

Few other studies [10, 350, 351] identified fixing patterns of violations of static good coding principles identified by tools such as `PMD` [12], `FindBugs` [352] and `Splint` [14]. However, in our study, we have studied real bug-fixing changes instead of coding principles' violations.

14.6.2 Identifying code-change patterns

Fluri et al. [353] identified code-change patterns in three Java applications using `ChangeDistiller`. Again, Martinez et al. [325] used `ChangeDistiller` to identify 18 code-change patterns. Molderez et al. [354] developed an automated system to mine code-change histories to detect unknown repetitive code-changes. Kim et al. [355, 356] discovered logical and structural changes at or above the method level. Breu and Zimmermann [357] extracted method call change patterns. Negara et al. [358] developed the tool `CodingTracker` and discovered previously unknown 10 code-change patterns. Kim et al. [316] developed the tool `LSdiff` that can group code-changes from systematic change patterns. While all these studies analyzed code-change patterns, we have studied bug-fixing changes by mining software repositories.

14.7 Summary

In this chapter, we have reported 38 bug-fixing *edit* patterns organized in 14 categories. This is the highest number of bug-fixing *edit* patterns identified in a single study. Using sequential pattern mining and clustering techniques, we have also exposed 37 new bug-fixing *nesting* patterns, which capture the locations of the bug-fixing edits within the nested code structure surrounding them. These new set of *nesting* patterns is a novel contribution that adds a new dimension to our understanding of bug-fixing patterns. The findings are derived from in-depth quantitative and qualitative analysis of 4,653 buggy revisions of five software systems written in Java.

Four of the identified bug-fixing *edit* patterns are completely new. They are related to `constructor`, `catch` and `enum` code constructs. The rest 34 of the identified bug-fixing *edit* patterns confirm those reported in earlier studies. Among the 14 identified categories of bug-fixing *edit* patterns, Method Call (MC), Method Declaration (MD), Local Variable (LV), If-related (IF), Assignment (AS) and Return (RT) are the six most dominant categories that frequently host bug-fixing edits. Four least dominant categories are Try (TY), Switch (SW), Catch (CA), and Enum (EN). These findings are in accordance with the observations reported in other studies [318, 327, 326].

The new set of 37 *nesting* patterns is divided in six categories and organized in descending order of their frequencies as follows: If-related (IF), Compound (COM), Loop (LP), Try-catch (TY-CA), Synchronize (SYN), and Chain Method Invocation (CMI). Our analysis of the *nesting* patterns reveals additional insights into bug-fix patterns. We have found that any nodes/blocks associated with `if` blocks are the most bug-prone. The *nesting* pattern "`if` block inside `loop` block" experience the highest number bug-fixing edits, followed by the "`if` block inside another `if` block" *nesting* pattern. Moreover, for the first time in this study, we have discovered that *nesting* patterns in CMI category experience a significant number of bug-fixing edits. Our analysis of the nesting patterns also indicates nodes/blocks inside `try-catch` and `synchronized` are bug-prone.

The findings from this work deepen our understanding of bug-fix patterns. Both the bug-fixing *edit* patterns and *nesting* patterns can also be useful in devising techniques for automated program repair. For example, existing patch generation algorithms can incorporate patterns of bug-fix edits and their locations in nested code structures to maximize the probability of success. Our future work will explore these possibilities.

Chapter 15

Conclusion

Now-a-days, software systems are integral parts of most of the technologies that people use. Since the increasing of uses of software systems, many instances of software failures have been causing tremendous losses in lives and dollars. Software systems mainly fail due to software bugs (i.e., faults and security issues). The fight against software bugs exists since software existed. Although many tools and techniques are devised and applied to identify, prevent, and minimizing software bugs, a numerous numbers of software bugs remain undetected and shipped with software systems. Thus, further studies are required to gain more insights into the causes of software bugs, devise new techniques, and strengthen existing ones.

In this thesis, we take on a holistic approach to mine and analyze human affects expressed in natural language texts (e.g., commit comments), and bug patterns (e.g., security vulnerabilities, code smells, and bug-fixing patterns) identified in source code to gain actionable insights that can be leveraged in minimizing bugs and improving quality and security of software systems.

The remaining of the chapter is organized as follows. In Section 15.1, we present a brief summary of the entire thesis. Section 15.2 points to the major contributions of this thesis. Section 15.3 discusses the limitations of this thesis. Finally, Section 15.4 concludes the chapter with future research directions.

15.1 Summary

In this thesis, we have mined and analyzed human affects (i.e., sentiments and emotions) and bug patterns (i.e., software bugs, code smells, and security vulnerabilities) from natural language textual artifacts and source code, respectively of software systems to gain actionable insights that can help in minimizing software bugs and improving software quality. To mine human developers' sentiments and emotions, we have developed three improved tools `SentiStrength-SE`, `DEVA`, and `MarValous`. Those tools are the first of their types that are capable of detecting sentiments and emotions at different levels in software engineering texts. While `SentiStrength-SE` is capable of detecting positivity and negativity of sentiments, `DEVA` and `MarValous` can detect four distinct emotional states, such as *excitement, stress, depression,* and *relaxation*. `MarValous` is developed using machine learning and natural language processing techniques that has achieved higher accuracy compared to `DEVA` that is developed using domain *dictionaries* and *sets of heuristics*. To evaluate `DEVA`, we have produced a benchmark dataset consisting of 1,795 JIRA issue comments manually annotated with the four emotional states identified in those comments. We have also compared the performance of `SentiStrenghth-SE` against its domain-specific counterparts and found that `SentiStrenghth-SE` has shown consistently higher recall value compare to other tools [188, 359].

Using a state-of-the-art tool, we have studied the developers' sentimental variations with respect to the underlying development tasks (e.g., bug-fixing, new feature implementation), development periods (i.e., days and times), teams' and projects' sizes. We expose opportunities to exploit developers' sentiments for higher productivity and improved software quality. In a separate study, the developers' emotional variations in the buggy and non-buggy commits are captured to derive insights into the role of sentimental variations of commits when they introduce new defects in source code. We have also conducted a survey to identify more roles and applications of sentiments and emotions in various software engineering activities.

While developers' sentiments can be leveraged in minimizing software bugs, their malpractices can lead to software bugs and degrade code quality. We have also mined and analyzed one of the dominant malpractices, i.e, copy-paste practice of developers to identify its detrimental impacts in software development. Recall that developers' such practices of copy-paste actions result in code clones. To gain insights into detrimental implications of code cloning, we have used static source code analysis techniques to mine *code smells* and *security vulnerabilities* in cloned and non-clone code. Our studies have revealed that the security vulnerabilities found in code clones have higher severity of security risks compared to those in non-cloned code. However, the proportion (i.e., density) of vulnerabilities in clones and the non-cloned code have not had any significant difference. We have also found no significant differences between the proportion of *smelly code* in code clones and clone-free source code. Surprisingly, among the three categories (i.e., *Type-1*, *pure Type-2*, and *pure Type-3*) of clones studied in our work, *Type-1* clones are found to be the most smelly whereas *pure Type-3* are the least. These findings inform developers about the characteristics, impacts, and implications of code cloning and problematic clones, which demand extra care. Such information also helps developers to minimize spreading of smelly code and security vulnerabilities that eventually can minimize software bugs and improve software quality.

We have also conducted an in-depth quantitative and qualitative analysis of buggy revisions in understanding of the common patterns of bug-fixing changes. In the study, we have identified not only existing but also new patterns of bug-fixing changes. We also identify nested bug-fix patterns, which is a novel contribution of this study. The findings of the study can be leveraged for avoiding common mistakes of developers that cause bugs and in developing automatic program repair tools.

15.2 Contributions

While we have provided the details of our research in the earlier chapters, and presented an abridged summary of the entire thesis in Section 15.1, here we outline the major contributions of this thesis as follows:

Detecting developers' sentiments and emotions: We have first presented an in-depth qualitative study to identify the difficulties in automated sentiment analysis in software engineering texts. We develop a number of rules and a domain specific dictionary to address some of the identified difficulties. Our new domain dictionary and the rules are integrated in `SentiStrength-SE`, a tool we have developed for improved sentiment analysis in textual contents in a technical domain, especially in software engineering. Our `SentiStrength-SE` is the first domain-specific sentiment analysis tool especially designed for software engineering text.

For the first time, we have conducted a comparative study of the performances of three software engineering domain-specific sentiment analysis tools (i.e., `SentiStrength-SE`, `Senti4SD`, and `SentiCR`) on three technical datasets. Our study has not found any clear-cut winner among the tools for sentiment analysis.

We have developed `DEVA`, a tool for automated emotion detection in software engineering texts. `DEVA` is unique from existing tools as it is capable of detecting both valence and arousal in text and mapping them for capturing individual emotional states (e.g., *excitement, stress, depression, relaxation,* and *neutrality*). `DEVA` applies a lexical approach with an arousal dictionary and a valence dictionary, both crafted for software engineering text. In addition, `DEVA` includes a set of heuristics, which help the tool to maintain high accuracy. To evaluate `DEVA`, we have also constructed a ground-truth dataset consisting of 1,795 JIRA issue comments.

To create an improved version of `DEVA`, we have developed another tool that is the first *Machine Learning (ML) based* tool for improved detection of four emotional states *excitation, stress, depression,* and *relaxation* expressed in software engineering texts. We have evaluated nine supervised ML algorithms incorporated in `MarValous` to measure their applicabilities in detecting the aforementioned four emotional states. We have created a unified dataset by combining two different benchmark datasets that can be reused in training and testing similar tools as we have done for our ML-based tool.

Understanding developers' sentiments: As software development is immensely affected by emotions, we have conducted a quantitative empirical study to understand developers' sentimental variations in different types of development activities (e.g., bug-fixing tasks) and development periods (i.e., days and times), and impacts of sentiments on software artifacts (i.e., commit messages).

We have found that developers express significantly more positive sentiment than negative sentiments in commits for *bug-fixing* and *refactoring* tasks. Surprisingly, the opposite scenario is found for *new feature implementation* tasks. We have also found significant positive correlation between the lengths of commit messages and the sentiment expressed in them. We have also distinguished a group of developers who express more positive sentiments at bug-fixing commit messages where as another group shows more negative sentiment. Such findings can be utilized for effective tasks' assignments and building cohesive teams.

In a separate study, we have quantitatively studied the sentimental variations in bug-introducing and bug-fixing commit messages. We have found that both bug-introducing and bug-fixing commit messages have overall statistically significantly higher positive sentimental scores compared to negative sentimental scores. We have also observed similar findings while analyzing sentimental scores in bug-introducing and bug-fixing commit messages with respect to three working periods (e.g., before, after, and during working hours). An exception is found where no significant difference found between positive and negative sentimental scores in bug-fixing commit messages posted during *after work hours*. Such variations of sentiments in commit messages have potential to be used in a classifier to predict bugs while developers commit code.

Understanding impacts of developers copy-paste action: We have conducted a comparative study on different types of clones and non-cloned code on the basis of their *code smells*, which may lead to software defects, vulnerabilities, and other issues in future. We have found that *Type-1* clones contain the most amount of smelly code whereas *pure Type-3* are the least. Moreover, we have identified five types of code smells that appear more frequently in cloned code compared to non-cloned code. We have also conducted a similar comparative study on different types of clones and non-cloned code on the basis of their *security vulnerabilities*. The latter study has revealed that the security vulnerabilities found in code clones have higher severity of security risks compared to those in non-cloned code. However, the proportion (i.e., density) of vulnerabilities in clones and non-cloned code does not have any significant difference. The findings from the previous two studies add to our understanding of the characteristics and impacts of clones, which will be useful in clone-aware software development with improved software security.

Understanding fixing patterns of software bugs: We have conducted an in-depth quantitative and qualitative analysis over 4,653 buggy revisions of five software systems. Our study has identified 38 bug-fixing *edit* patterns and exposes 37 new patterns of nested code structures, which frequently host the bug-fixing edits. While some of the *edit* patterns were reported in earlier studies, these *nesting* patterns are *new* and were never targeted before. Findings of the study advance our understanding of patterns and nested locations of software bug-fixings, i.e., bugs. An understanding of the common patterns of bug-fixing changes is useful in several ways: (a) such knowledge can help developers in proactively avoiding coding patterns that lead to bugs and (b) bug-fixing patterns can be exploited in devising techniques for automatic program repair.

15.3 Limitations

We have tested our developed tools (i.e., `SentiStrength-SE`, `DEVA`, and `MarValous`) using limited number of benchmark datasets, which is an obvious limitation of this thesis. The tools could be more generalizable if we could use enough number of datasets to test the tools' external validity. However, this was not possible, since the number of benchmark datasets was limited during the development of the tools [52, 110]. The detection of irony, sarcasm, and subtle sentiments and emotions expressed in text is still a challenging task for the developed tools. Uncommon scenarios, such as the use of attenuators (e.g., rather, lack) and neutralizers (e.g., however, although) [360] in sentences may hamper the performance of the tools.

In general, the empirical studies presented in this dissertation might have suffered from the limitations of the used tools and algorithms used for data collection and analysis. Although in the studies, we included a large number of subject systems collected from diverse application domains and written in different programming languages, those subject systems were mostly open-source. This can be regarded as a limitation of this thesis. One may question, to what extent the findings and clone management implications derived from those studies can be expected to hold for industrial software systems.

15.4 Future Research Directions

1. **Tool Development for Capturing Human Affects**:

    (a) **Scopes for tool improvement**: As we discussed in Chapter 4 and mentioned in Section 15.3, to improve tools' accuracies, the research community needs to develop approaches for detecting irony, sarcasm, and subtle sentiments and emotions expressed in software engineering texts. We also need better techniques to handle negation, attenuators, and neutralizers in sentences that play critical roles in detecting human affects. Although we have applied a technique to to separate natural language from technical content (e.g., source code, stack-overflow and HTML elements), there is a scope for improvement in that area.

    Moreover, If a sentence expresses both positive and negative sentiments, the machine learning based tools (e.g., Senti4SD [128], SentiCR [162], and EmoTxt [56]) can detect either positive or negative sentiment, which is major limitation of those tools. Moreover, those machine learning based tools can not detect intensity (i.e., level) of sentiments or emotions. Thus, a machine learning based tool is required that can detect both sentiments and their intensities from a sentence.

   (b) **New tool development**: In technical and social forums developers often discuss around software-specific entities (e.g., tools, libraries, and APIs) where developers provide their valuable opinions on the various aspects (e.g., bug, performance, documentation, and security) of the entities. Such opinions often sentimentally polarized (i.e., positive or negative) that play a pivotal role to a considerable degree on the perceptions of other developers about those entities. Such perceptions influence the choices that developers make about whether and how they should use those entities. To automatically mine the perceptions from developers' discussions in a meaningful way, we need to combine three things: (i) entity, is an object about what the opinion is expressed, (ii) aspect, is the property of an entity, and (iii) affects (e.g., sentiments and emotions) expressed in opinions. However, the research community lacks a tool that can combine those mentioned three things.

2. **Creation of Benchmark Dataset**:

   (a) **A uniform guideline to annotate human affects**: Studies [361, 189] found that humans are inconsistent in identifying affects in developers' comments/discussions. This demonstrates the need for *standardized annotation schemes* for the human annotators to build a benchmark dataset, and then perform training on the tools to perform reliable affect analysis in the software engineering domain.

   (b) **More benchmark dataset is required**: To the best of our knowledge, there are only five benchmark datasets [130, 52, 162, 362] that consist of only 11,233 labeled comments. Among the labeled comments around 40% comments are neutral that express no affect. The scarce of labeled data can cover only very limited expressions and thus can not guarantee that tools trained and tested on those datasets will be generalized. To address such a problem, we need to create more benchmark datasets that contain a higher number of affective comments. We also need a publicly available large dataset where comments will be annotated to identify entity, aspect, affect, and dependency among those to train and test a tool for entity- and aspect-level affect analysis.

3. **Future Application of Human Affects**:

   (a) **Requirement analysis**:

   As discussed in Chapter 10, researchers [257, 258, 259, 261] mined sentimental opinions of non-technical users to identify new requirements and various types of maintenance requests of different applications. We can also mine and analyze developers' sentimental discussions and opinions to identify developers' new requirements and maintenance requests regarding various APIs, tools, and libraries they use.

(b) **Predictive analysis**: Garcia et al. [42] analyzed the relation between the emotions and the activity of developers in the Open Source Software project Gentoo, and developed an emotion based predictor to predict the risk of developers leaving the project. In future, research community can plan to develop a predictor to predict whether the developers' will stay in or leave their projects by analyzing developers' affects in comments.

(c) **Effective communication and collaboration**: It would be very interesting to see the impacts of affective communications on code review and pull-request based development. More studies are required to analyze affects of discussions in issue repositories to see their impacts on issue-fixings. Developing a tool to detect the tone/affects of an email while sending that can contribute in effective communication and collaboration.

(d) **Task Assignment**: In Chapter 8, we have seen that developers have their preferences of doing different types of jobs (e.g., bug-fixing and new feature development). In future, Knowing such preferences of developers can be utilized to assign them their preferred jobs. Moreover, we will verify whether task assignments based on preferences resulted in high quality of end product or not.

4. **Studies Required to Discover More Bug-fixing Patterns**: In Chapter 14, we have conducted a study to identify bug-fix patterns in five software systems. The research community needs to conduct large studies to identify more unknown bug-fix patterns to reap the full benefits of it to develop approaches for automatic program repair. Moreover, studies should be conducted by focusing on specific bug or issue types, such as security bugs, performance issues, and client-side programming (e.g., Javascript) mistakes.

Bibliography

[1] G. Walia, J. Carver, Using error information to improve software quality, in: Proceedings of the IEEE International Symposium on Software Reliability Engineering, 2013, pp. 107–107.

[2] F. Huang, B. Liu, Software defect prevention based on human error theories, Chinese Journal of Aeronautics 30 (3) 1054–1070.

[3] D. Graziotin, X. Wang, P. Abrahamsson, Do feelings matter? On the correlation of affects and the self-assessed productivity in software engineering, J. of Softw.: Evolution and Proc. 27 (7) (2015) 467–487.

[4] E. Guzman, B. Bruegge, Towards emotional awareness in software development teams, in: Proceedings of the International Symposium on the Foundations of Software Engineering, 2013, pp. 671–674.

[5] D. Amalfitano, A. Fasolino, P. Tramontana, B. Ta, A. Memon, Mobiguitar: Automated model-based testing of mobile apps, IEEE Software 32 (5) (2015) 53–59.

[6] D. Amalfitano, A. Fasolino, P. Tramontana, S. Carmine, A. Memon, Using gui ripping for automated testing of android applications, in: Proceedings of the International Conference on Automated Software Engineering, 2012, pp. 258–261.

[7] V. Garousi, Empirical analysis of a genetic algorithm-based stress test technique, in: Proceedings of the 10th annual conference on Genetic and evolutionary computation, 2008, pp. 1743–1750.

[8] Z. Jiang, A. Hassan, A survey on load testing of large-scale software systems, IEEE Transactions on Software Engineering 41 (11) (2015) 1091–1117.

[9] Flawfinder - C/C++ Source Code Analyzer, http://www.dwheeler.com/flawfinder/, July 2017.

[10] K. Liu, D. Kim, T. Bissyande, S. Yoo, Y. Traon, Mining fix patterns for findbugs violations, IEEE Transactions on Software Engineering (2018) 1–24.

[11] T. Kamiya, S. Kusumoto, K. Inoue, CCFinder: a multilinguistic token-based code clone detection system for large scale source code, IEEE Trans. Softw. Eng. 28 (7) (2002) 654–670.

[12] PMD - Source Code Analyzer, http://pmd.sourceforge.net, July 2017.

[13] Cppcheck - A tool for static C/C++ code analysis, http://cppcheck.sourceforge.net, July 2017.

[14] SPLINT - Secure Programming LINT, http://www.splint.org, July 2017.

[15] G. Baah, A. Podgurski, M. Harrold, Causal inference for statistical fault localization, in: Proceedings of the International symposium on Software testing and analysis, 2010, pp. 73–84.

[16] E. Alves, M. Gligoric, V. Jagannath, M. d'Amorim, Fault-localization using dynamic slicing and change impact analysis, in: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, 2011, pp. 520–523.

[17] R. Abreu, P. Zoeteweij, R. Golsteijn, A. Gemund, A practical evaluation of spectrum-based fault localization, Journal of Systems and Software 82 (11) (2009) 1780–1792.

[18] R. Abreu, A. Gonzalez, P. Zoeteweij, A.Gemund, Automatic software fault localization using generic program invariants, in: Proceedings of the ACM Symposium in Applied Computing, 2008, pp. 712–717.

[19] M. Martinez, M. Monperrus, Astor: A program repair library for java, in: Proceedings of the International Symposium on Software Testing and Analysis, 2016, pp. 441–444.

[20] Y. Xiong, J. Wang, R. Yan, J. Z. S. Han, G. H. L. Zhang, Precise condition synthesis for program repair, in: Proceedings of the Proceedings of the International Conference on Software Engineering, 2017, pp. 416–426.

[21] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, X. Chen, Shaping program repair space with existing patches and similar code, in: Proceedings of the International Symposium on Software Testing and Analysis, 2018, pp. 298–309.

[22] B. Liblit, A. Aiken, A. Zheng, M. Jordan, Bug isolation via remote program sampling, in: Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, 2003, pp. 141–154.

[23] M. Harman, Automated patching techniques: the fix is in: technical perspective, Communications of the ACM 53 (5) (2010) 108–108.

[24] J. Reason, Human error: models and management, West J Med. 172 (6) (2000) 393–396.

[25] F. Rahman, C. Bird, P. Devanbu, Clones: what is that smell?, in: Proceedings of the International Conference on Mining Software Repositories, 2010, pp. 72–81.

[26] M. Choudhury, S. Counts, Understanding affect in the workplace via social media, in: Proceedings of the Computer supported cooperative work, 2013, pp. 303–316.

[27] R. Feldt, L. Angelis, R. Torkara, M. Samuelssonc, Links between the personalities, views and attitudes of software engineers, Information and Software Technology 52 (6) (2010) 611–624.

[28] R. Palacios, A. López, A. Crespo, P. Acosta, A study of emotions in requirements engineering, Organizational, Business, and Technological Aspects of the Knowledge Society 112 (2010) 1–7.

[29] P. Dewan, Towards emotion-based collaborative software engineering, in: Proceedings of the International Workshop on Cooperative and Human Aspects of Software Engineering, 2015, pp. 109–112.

[30] P. Denning, Moods, Communications of the ACM 55 (12) (2012) 33–35.

[31] M. Wrobel, Emotions in the software development process, in: Proceedings of the International Conference on Human System Interaction, 2013, pp. 518–523.

[32] D. McDuff, A. Karlson, A. Kapoor, A. Roseway, M. Czerwinski, Affectaura: an intelligent system for emotional memory, in: Proceedings of the Conference on Human Factors in Computing Systems, 2012, pp. 849–858.

[33] G. Destefanis, M. Ortu, S. Counsell, M. Marchesi, R. Tonelli, Software development: do good manners matter?, PeerJ PrePrints (2015) 1–17.

[34] E. Guzman, D. Azócar, Y. Li, Sentiment analysis of commit comments in github: An empirical study, in: Proceedings of the International Conference on Mining Software Repositories, 2014, pp. 352–355.

[35] F. Calefato, F. Lanubile, Affective trust as a predictor of successful collaboration in distributed software projects, in: Proceedings of the International Workshop on Emotion Awareness in Software Engineering, 2016, pp. 3–5.

[36] S. Chowdhury, A. Hindle, Characterizing energy-aware software projects: Are they different?, in: Proceedings of the International Conference on Mining Software Repositories, 2016, pp. 508–511.

[37] M. Islam, M. Zibran, Exploration and exploitation of developers' sentimental variations in software engineering, International Journal of Software Innovation 4 (4) (2016) 35–55.

[38] M. Islam, M. Zibran, Towards understanding and exploiting developers' emotional variations in software engineering, in: Proceedings of the International Conference on Software Engineering Research Management and Applications, 2016, pp. 185–192.

[39] M. Mantyla, B. Adams, G. Destefanis, D. Graziotin, M. Ortu, Mining valence, arousal, and dominance – possibilities for detecting burnout and productivity, in: Proceedings of the International Conference on Mining Software Repositories, 2016, pp. 247–258.

[40] M. Ortu, B. Adams, G. Destefanis, P. Tourani, M. Marchesi, R. Tonelli, Are bullies more productive? Empirical study of affectiveness vs. issue fixing time, in: Proceedings of the International Conference on Mining Software Repositories, 2015, pp. 303–313.

[41] D. Pletea, B. Vasilescu, A. Serebrenik, Security and emotion: Sentiment analysis of security discussions on github, in: Proceedings of the International Conference on Mining Software Repositories, 2014, pp. 348–351.

[42] D. Garcia, M. Zanetti, F. Schweitzer, The role of emotions in contributors activity: A case study on the gentoo community, in: Proceedings of the International Conference on Cloud and Green Computing, 2013, pp. 410–417.

[43] P. Tourani, Y. Jiang, B. Adams, Monitoring sentiment in open source mailing lists – exploratory study on the apache ecosystem, in: Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research, 2014, pp. 34–44.

[44] N. Novielli, F. Calefato, F. Lanubile, Towards discovering the role of emotions in stack overflow, in: Proceedings of the International Workshop on Social Software Engineering, 2014, pp. 33–40.

[45] M. Thelwall, K. Buckley, G. Paltoglou, Sentiment strength detection for the social web, Journal of the American Society for Info. Science and Tech. 63(1) (2012) 163–173.

[46] NLTK, Natural Language Toolkit for Sentiment Analysis,

http://www.nltk.org/api/nltk.sentiment.html, last access: March 2019.

[47] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. Manning, A. Ng, C. Potts, Recursive deep models for semantic compositionality over a sentiment treebank, in: Proceedings of the Conference on Empirical Methods in Natural Language Processing, 2013, pp. 1631–1641.

[48] R. Jongeling, S. Datta, A. Serebrenik, Choosing your weapons: On sentiment analysis tools for software engineering research, in: Proceedings of the International Conference on Software Maintenance and Evolution, 2015, pp. 531–535.

[49] N. Novielli, List of Tools Used in Software Engineering to Detect Emotions, http://www.slideshare.net/nolli82/the-challenges-of-affect-detection-in-the-social-programmer-ecosystem, last access: March 2019.

[50] N. Novielli, F. Calefato, F. Lanubile, The challenges of sentiment detection in the social programmer ecosystem, in: Proceedings of the International Workshop on Social Software Engineering, 2015, pp. 33–40.

[51] P. Tourani, B. Adams, The impact of human discussions on just-in-time quality assurance, in: Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering, 2016, pp. 189–200.

[52] M. Ortu, A. Murgia, G. Destefanis, P. Tourani, R. Tonelli, M. Marchesi, B. Adams, The emotional side of software developers in JIRA, in: Proceedings of the International Conference on Mining Software Repositories, 2016, pp. 480–483.

[53] V. Sinha, A. Lazar, B. Sahrif, Analyzing developer sentiment in commit logs, in: Proceedings of the International Conference on Mining Software Repositories, 2016, pp. 520–523.

[54] SentiStregth-SE, Sentiment Analysis Tool, freely available for download, http://laser.cs.uno.edu/Projects/Projects.html, last access: March 2019.

[55] F. Calefato, F. Lanubile, F. Maiorano, N. Novielli, Sentiment polarity detection for software development, Empirical Software Engineering (2017) 352–1382.

[56] F. Calefato, F. Lanubile, N. Novielli, Emotxt: A toolkit for emotion recognition from text, in: Proceedings of the Seventh International Conference on Affective Computing and Intelligent Interaction Workshops and Demos, 2017.

[57] M. Islam, M. Zibran, A comparison of dictionary building methods for sentiment analysis in software engineering text, in: Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 2017, pp. 478–479.

[58] M. Islam, M. Zibran, Deva: Sensing emotions in the valence arousal space in software engineering texmotions in the valence arousal space in software engineering text, in: proceedings of the 33rd ACM/SIGAPP Symposium On Applied Computing (SAC), 2018, pp. 1536 –1543.

[59] URL for downloading DEVA and Benchmark Dataset, https://figshare.com/s/277026f0686f7685b79e, verified: Dec 2017.

[60] D. Graziotin, X. Wang, P. Abrahamsson, Are happy developers more productive? The corre-

lation of affective states of software developers and their self-assessed productivity, in: Proceedings of the International Conference on Product-Focused Software Process Improvement, 2013, pp. 50–64.

[61] I. Khan, W. Brinkman, R. Hierons, Do moods affect programmers' debug performance?, Cogn. Technol. Work 13 (4) (2010) 245–258.

[62] T. Lesiuk, The effect of music listening on work performance, Psychology of Music 33 (2) (2005) 173–191.

[63] A. Murgia, P. Tourani, B. Adams, M. Ortu, Do developers feel emotions? An exploratory analysis of emotions in software artifacts, in: Proceedings of the International Conference on Mining Software Repositories, 2014, pp. 261–271.

[64] T. Shaw, The emotions of systems developers: an empirical study of affective events theory, in: SIGMIS CPR, 2004, pp. 124–126.

[65] M. Islam, M. Zibran, Sentiment analysis of software bug related commit messages, in: Proceedings of the 27th International Conference on Software Engineering and Data Engineering, 2018.

[66] M. Rieger, Effective clone detection without language barriers, PhD thesis, Institut fuř Informatik und angewandte Mathematik, Germany (2005).

[67] C. Roy, J. Cordy, A survey on software clone detection research, Tech Report TR 2007-541, Queens University, Canada (2007).

[68] M. Zibran, R. Saha, M. Asaduzzaman, C. Roy, Analyzing and forecasting near-miss clones in evolving software: An empirical study, in: Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems, 2011, pp. 295–304.

[69] M. Rieger, S. Ducasse, M. Lanza, Insights into system-wide code duplication, in: Proceedings of the Working Conference on Reverse Engineering, 2004, pp. 100–109.

[70] S. Ducasse, M. Rieger, S. Demeyer, A language independent approach for detecting duplicated code, in: Proceedings of the IEEE International Conference on Software Maintenance, 1999, pp. 109 –118.

[71] C. Kapser, M. Godfrey, "Cloning considered harmful" considered harmful: patterns of cloning in software, Empirical Software Engineering 13 (2008) 645–692.

[72] E. Juergens, F. Deissenboeck, B. Hummel, S. Wagner, Do code clones matter?, in: Proceedings of the International Conference on Software Engineering, 2009, pp. 485–495.

[73] A. Lozano, M. Wermelinger, Assessing the effect of clones on changeability, in: Proceedings of the IEEE International Conference on Software Maintenance, 2008, pp. 227–236.

[74] M. Fowler, K. Beck, J.Brant, W. Opdyke, D. Roberts, Refactoring: Improving the Design of Existing Code, Addison Wesley, 1999.

[75] M. Zibran, C. Roy, Conflict-aware optimal scheduling of code clone refactoring, IET Software 7 (3) (2013) 167–186.

[76] R. T. Institute, The economic impacts of inadequate infrastructure of software testing, RTI Project Report 7007.011, National Inst. of Standards and Tech. (2002).

[77] I. Baxter, M. Conradt, J. Cordy, R. Koschke, Software clone management towards industrial application (dagstuhl seminar 12071), DagStuhl Report 2 (2) (2012) 21–57.

[78] N. Göde, J. Harder, Clone stability, in: Proceedings of the European Conference on Software Maintenance and Reengineering, 2011, pp. 65–74.

[79] K. Hotta, Y. Sano, Y. Higo, S. Kusumoto, Is duplicate code more frequently modified than non-duplicate code in softw. evolution?: an emp. study on open source softw., in: IWPSE-EVOL, 2010, pp. 73–82.

[80] J. Krinke, Is cloned code more stable than non-cloned code?, in: Proceedings of the Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation, 2008, pp. 57–66.

[81] J. Krinke, Is cloned code older than non-cloned code?, in: Proceedings of the International Workshop on Software Clone, 2011, pp. 28–33.

[82] M. Mondal, C. Roy, K. Schneider, An empirical study on clone stability, ACM Applied Computing Review 12 (3) (2012) 20–36.

[83] L. Barbour, F. Khomh, Y. Zou, Late propagation in software clones, in: Proceedings of the IEEE International Conference on Software Maintenance, 2011, pp. 273–282.

[84] K. Inoue, Y. Higo, N. Yoshida, E. Choi, S. Kusumoto, K. Kim, W. Park, E. Lee, Experience of finding inconsistently-changed bugs in code clones of mobile software, in: Proceedings of the International Workshop on Software Clone, 2012, pp. 94–95.

[85] L. Jiang, Z. Su, E. Chiu, Context-based detection of clone-related bugs, in: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, pp. 55–64.

[86] E. Juergens, B. Hummel, F. Deissenboeck, M. Feilkas, Static bug detection through analysis of inconsistent clones, in: TESO, 2008, pp. 443–446.

[87] J. Li, M. Ernst, CBCD: Cloned buggy code detector, in: Proceedings of the International Conference on Software Engineering, 2012, pp. 310–320.

[88] G. Selim, L. Barbour, W. Shang, B. Adams, A. Hassan, Y. Zou, Studying the impact of clones on software defects, in: Proceedings of the Working Conference on Reverse Engineering, 2010, pp. 13–21.

[89] S. Xie, F. Khomh, Y. Zou, An empirical study of the fault-proneness of clone mutation and clone migration, in: Proceedings of the International Conference on Mining Software Repositories, 2013, pp. 149–158.

[90] N. Göde, R. Koschke, Frequency and risks of changes to clones, in: Proceedings of the International Conference on Software Engineering, 2011, pp. 311–320.

[91] A. Lozano, M. Wermelinger, B. Nuseibeh, Evaluating the harmfulness of cloning: A change

based exp., in: Proceedings of the International Conference on Mining Software Repositories, 2007, pp. 18–21.

[92] J. Jang, A. Agrawal, D. Brumley, Redebug: Finding unpatched code clones in entire os distributions, in: SSP, 2012, pp. 48–62.

[93] H. Li, H. Kwon, J. Kwon, H. Lee, CLORIFI: software vulnerability discovery using code clone verification, Concurrency and Computation: Practice and Experience 28 (6) (2015) 1900–1917.

[94] R. Tonder, C. Goues, Defending against the attack of the micro-clones, in: Proceedings of the International Conference on Program Comprehension, 2016, pp. 1–4.

[95] M. Islam, M. Zibran, A comparative study on vulnerabilities in categories of clones and non-cloned code, in: Proceedings of the International Workshop on Software Clone, 2016, pp. 8–14.

[96] M. Islam, M. Zibran, A. Nagpal, Security vulnerabilities in categories of clones and non-cloned code: An empirical study, in: Proceedings of the International Symposium on Empirical Software Engineering and Measurement, 2017, pp. 20–29.

[97] M. Islam, M. Zibran, On the characteristics of buggy code clones: A code quality perspective, in: Proceedings of the International Workshop on Software Clones, 2018, pp. 23 – 29.

[98] E. Campos, M. Maia, Common bug-fix patterns: A large-scale observational study, in: Proceedings of the Empirical Software Enginerign and Measurement, 2017, pp. 404–413.

[99] Merriam-Webster Online, http:// www.merriam-webster.com/dictionary/, last access: March 2019.

[100] P. Lang, M. Greenwald, M. Bradley, A. Hamm, Looking at pictures: A ective, facial, visceral, and behavioral reactions, Psychophysiology 30 (3) (1993) 261–273.

[101] M. Munezero, C. Montero, Are they different? affect, feeling, emotion, sentiment, and opinion detection in text, IEEE Transaction on Affective Computing 5 (2) (2014) 101–111.

[102] E. Shouse, Feeling, emotion, affect, Journal of Media and Culture 8 (6) (2005) NA.

[103] R. Lane, P. Chua, R. Dolan, Common e ects of emotional valence, arousal and attention on neural activation during visual processing of pictures, Neuropsychologia 37 (9) (1999) 989–997.

[104] M. Zajenkowski, E. Goryoska, M. Winiewski, Variability of the relationship between personality and mood, Personality and Individual Differences 52 (7) (2012) 858–861.

[105] M. Hasan, E. Rundensteiner, E. Agu, Emotex: Detecting emotions in twitter messages, in: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, 2014, pp. 27–31.

[106] J. Russell, A. Mehrabian, Evidence for a three-factor theory of emotions, Journal of Research in Personality 11 (3) (1977) 273–294.

[107] R. Palacios, C. Lumbreras, P. Acosta, A. Acosta, Using the affect grid to measure emotions

in software requirements engineering, Journal of Universal Computer Science 17 (9) (2011) 1281–1298.

[108] Common Weakness Enumeration, https://cwe.mitre.org, July 2017.

[109] M. Islam, M. Zibran, SentiStrength-SE: Exploiting domain specificity for improved sentiment analysis in software engineering text, Journal of System and Software 145 (2018) 125–146.

[110] M. Islam, M. Zibran, Leveraging automated sentiment analysis in software engineering, in: Proceedings of the Mining Software Repositories, 2017, pp. 203–214.

[111] T. Ahmed, A. Bosu, A. Iqbal, S. Rahimi, Senticr: a customized sentiment analysis tool for code review interactions, in: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, 2017, pp. 106–111.

[112] J. Bross, H. Ehrig, Automatic construction of domain and aspect specific sentiment lexicons for customer review mining, in: CIKM, 2013, pp. 1077–1086.

[113] H. Hammer, A. Yazidi, A. Bai, P. Engelstad, Building domain specific sentiment lexicons combining information from many sentiment lexicons and a domain specific corpus, in: Proceedings of the Computer Science and Its Applications, 2015, pp. 205–216.

[114] M. Kim, J. Kim, C. Juing, Performance evaluation of domain-specific sentiment dictionary construction methods for opinion mining, Intl. J. of Database Theory and Application 9 (8) (2016) 257–268.

[115] L. Young, S. Soroka, Affective news: The automated coding of sentiment in political texts, Political Communication 29 (2012) 205–231.

[116] A. Reagan, B. Tivnan, J. J. Williams, C. Danforth, P. Dodds, Benchmarking senti. anal. methods for large-scale texts: A case for using continuum-scored words and word shift graphs, ArXiv e-printsarXiv:1512.00531.

[117] A. Pablos, M. Cuadros, G. Rigau, A comparison of domain-based word polarity estimation using different word embeddings, in: Proceedings of the International Conference on Language Resources and Evaluation, 2016, pp. 54–60.

[118] F. Nielsen, A new ANEW: Evaluation of a word list for sentiment analysis in microblogs, in: Proceedings of the ESWC 2011 Workshop on 'Making Sense of Microposts', 2011, pp. 93–98.

[119] T. Wilson, J. Wiebe, P. Hoffmann., Recognizing contextual polarity: An exploration of features for phrase-level sentiment analysis, Journal of Computational Linguistics 35 (3) (2009) 399–433.

[120] C. Hutto, E. Gilbert, Vader: A parsimonious rule-based model for sentiment analysis of social media text, in: Proceedings of the Proceedings of the Eighth International AAAI Conference on Weblogs and Social Media, 2014, pp. 216–225.

[121] S. Muller, T. Fritz, Stuck and frustrated or in flow and happy: Sensing developers' emotions and progress, in: Proceedings of the International Conference on Software Engineering, 2015,

pp. 688–699.

[122] M. Thelwall, Tensistrength: stress and relaxation magnitude detection for social media texts, Information Processing and Management 53 (1) (2017) 106–121.

[123] A. Bifet, E. Frank, Sentiment knowledge discovery in twitter streaming data, in: Proceedings of the International Conference on Discovery Science, 2010, pp. 1–15.

[124] M. Pennacchiotti, A. P. Democrats, Republicans and starbucks afficionados: User classification in twitter, in: Proceedings of the International Conference on Knowledge Discovery and Data Mining, 2011, pp. 430–438.

[125] B. Panga, L. Lee, S. Vaithyanathan, Thumbs up? Sentiment classification using machine learning techniques, in: Proceedings of the Conference on Empirical Methods in Natural Language Processing, 2002, pp. 79–86.

[126] Y. Wan, D. Gao, An ensemble sentiment classification system of twitter data for airline services analysis, in: Proceedings of the IEEE 15th International Conference on Data Mining Workshops, 2015, pp. 1318–1325.

[127] D. Alboaneen, H. Tianfield, Y. Zhang, Sentiment analysis via multi-layer perceptron trained by meta-heuristic optimisation, in: Proceedings of the IEEE International Conference on Big Data, 2017, pp. 4630–4635.

[128] F. Calefato, F. Lanubile, F. Maiorano, N. Novielli, Sentiment polarity detection for software development, Empirical Software Engineering (2017) 1–31.

[129] M. Islam, M. Zibran, Deva: Sensing emotions in the valence arousal space in software engineering text, in: Proceedings of the The ACM/SIGAPP Symposium On Applied Computing, 2018, pp. 1536–1543.

[130] N. Novielli, F. Calefato, F. Lanubile, A gold standard for emotion annotation in stack overflow, in: Proceedings of the International Conference on Mining Software Repositories, 2018, pp. 14–17.

[131] R. Dyer, H. Nguyen, H. Rajan, T. Nguyen, Boa: A language and infrastructure for analyzing ultra-large-scale software repositories, in: Proceedings of the International Conference on Software Engineering, 2013, pp. 422–431.

[132] F. Ramsey, D. Schafer, The Statistical Sleuth, 2nd Edition, Duxbury-Thomson Learning, 2002.

[133] A. Ciurumelea, A. Schaufelbuhl, S. Panichella, H. Gall, Analyzing reviews and code of mobile apps for better release planning, in: Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering, 2017, pp. 91–102.

[134] F. Calefato, F. Lanubile, M. Marasciulo, N. Novielli, Mining successful answers in stack overflow, in: Proceedings of the International Conference on Mining Software Repositories, 2015, pp. 430–433.

[135] F. Calefato, F. Lanubile, N. Novielli, Moving to stack overflow: Best-answer prediction in

legacy developer forums, in: Proceedings of the International Symposium on Empirical Software Engineering and Measurement, 2016.

[136] J. Jiarpakdee, A. Ihara, K. Matsumoto, Understanding question quality through affective aspect in Q&A site, in: Proceedings of the International Workshop on Emotion Awareness in Software Engineering, 2016, pp. 12–17.

[137] G. Yang, S. Baek, J. Lee, B. Lee, Analyzing emotion words to predict severity of software bugs: A case study of open source projects, in: Proceedings of the The ACM/SIGAPP Symposium On Applied Computing, 2017, pp. 1280–1287.

[138] C. Roy, J. Cordy, NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization, in: Proceedings of the International Conference on Program Comprehension, 2008, pp. 172–181.

[139] SourceMeter: Static source code analysis solution for Java, C/C++, C#, Python and RPG, https://www.sourcemeter.com, verified: Jan 2018.

[140] M. Wrobel, Towards the participant observation of emotions in software development teams, in: Proceedings of the Federated Conference on Computer Science and Information Systems, 2016, pp. 1545–1548.

[141] Gold Standard Dataset Labeled with Manually Annotated Emotions, http://ansymore.uantwerpen.be/system/files/uploads/artefacts/alessandro/MSR16/archive3.zip, last access: March 2019.

[142] R. Jongeling, P. Sarkar, S. Datta, A. Serebrenik, On negative results when using sentiment analysis tools for software engineering research, Empirical Software Engineering (2017) 1–42.

[143] N. Godbole, M. Srinivasaiah, S. Skiena, Large-scale sentiment analysis for news and blogs, in: Procceding of the First International AAAI Conference on Weblogs and Social Media, 2007.

[144] M. Hu, B. Liu, Mining and summarizing customer reviews, in: Proceedings of the International Conference on Knowledge Discovery and Data Mining, 2004, pp. 168–177.

[145] G. Qiu, B. Liu, J. Bu, C. Chen, Expanding domain sentiment lexicon through double propagation., in: Proceedings of the International Jont Conference on Artifical Intelligence, 2009, pp. 1199–1204.

[146] StanfordCoreNLP, Stanford Core NLP Sentiment Annotator, http://stanfordnlp.github.io/CoreNLP/sentiment.html, last access: March 2019.

[147] E. Riloff, A. Qadir, P. Surve, L. Silva, N. Gilbert, R. Huang, Sarcasm as contrast between a positive sentiment and negative situation, in: Proceedings of the Conference on Empirical Methods in Natural Language Processing, 2013, pp. 704–714.

[148] Q. Gan, Y. Yu, Restaurant rating: Industrial standard and word-of-mouth a text mining and multi-dimensional sentiment analysis, in: Proceedings of the Hawaii International Conference

on System Sciences, 2015, pp. 1332–1340.

[149] F. Koto, M. Adriani, A comparative study on twitter sentiment analysis: Which features are good?, in: Proceedings of the International Conference on Applications of Natural Language to Information Systems, 2015, pp. 453–457.

[150] J. Fleiss, Measuring nominal scale agreement among many raters, Psychological bulletin 76 (5) (1971) 378.

[151] N. Bettenburg, B. Adams, A. Hassan, A lightweight approach to uncover technical information in unstructured data, in: Proceedings of the International Conference of Program Comprehension, 2011, pp. 185–188.

[152] Jazzy- The Java Open Source Spell Checker, http://jazzy.sourceforge.net, last access: March 2019.

[153] T. Dietterich, Approximate statistical tests for comparing supervised classification learning algorithms, Journal of Neural Computation 10 (7) (1998) 1895–1923.

[154] M. Rahman, C. Roy, I. Keivanloo, Recommending insightful comments for source code using crowdsourced knowledge, in: Proceedings of the International Working Conference on Source Code Analysis and Manipulation, 2015, pp. 81–90.

[155] V. Sinha, Sentiment analysis on java source code in large sofyware repositories, Master's thesis, Youngstown State University, USA (2016).

[156] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. Manning, A. Ng, C. Potts, Recursive deep models for semantic compositionality over a sentiment treebank, in: Proceedings of the Conference on Empirical Methods in Natural Language Processing, 2013, pp. 1631–1642.

[157] SentiStregth-SE, Automatic Domain Independent Tool for Sentiment Analysis, http://sentistrength.wlv.ac.uk, last access: March 2019.

[158] S. Baccianella, A. Esuli, F. Sebastiani, Sentiwordnet 3.0: An enhanced lexical resource for sentiment analysis and opinion mining, in: Procceding of the International Conference on Language Resources and Evaluation, 2010, pp. 2200–2204.

[159] M. G. L. Gatti, M. Turchi, Sentiwords: Deriving a high precision and high coverage lexicon for sentiment analysis, IEEE Transactions on Affective Computing 7 (4) (2016) 409–421.

[160] A. Warriner, V. Kuperman, M. Brysbaert, Norms of valence, arousal, and dominance for 13,915 english lemmas, Behavior research methods 45 (4) (2013) 1191–1207.

[161] M. Mäntylä, N. Novielli, F. Lanubile, M. Claes, M. Kuutila, Bootstrapping a lexicon for emotional arousal in software engineering, in: Proceedings of the International Conference on Mining Software Repositories, 2017, pp. 1–5.

[162] T. Ahmed, A. Bosu, A. Iqbal, S. Rahimi, Senticr: a customized sentiment analysis tool for code review interactions, in: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, 2017, pp. 106–111.

[163] C. Blaz, K. Becker, Sentiment analysis in tickets for it support, in: Proceedings of the Inter-

national Conference on Mining Software Repositories, 2016, pp. 235–246.

[164] A. Rousinopoulos, G. Robles, J. Barahona, Sentiment analysis of free/open source developers: preliminary findings from a case study, Revista Eletronica de Sistemas de Informacao 13 (2) (2014) 1–21.

[165] E. Dragut, C. Yu, P. Sistla, W. Meng, Construction of a sentimental word dictionary, in: Proceedings of the International Conference on Information and Knowledge Management, 2010, pp. 1761–1764.

[166] L. Passaro, L. Pollacci, A. Lenci, Item: A vector space model to bootstrap an italian emotive lexicon, in: Proceedings of the Second Italian Conference on Computational Linguistics CLiC-it, 2015, pp. 215–220.

[167] M. Host, B. Regnell, C. Wohlin, Using students as subjects: A comparative study of students and professionals in lead-time impact assessment, Empirical Software Engineering 5 (3).

[168] S. Panichella, A. Sorbo, E. Guzman, C. Visaggio, G. Canfora, H. Gall, How can i improve my app? Classifying user reviews for software maintenance and evolutio, in: Proceedings of the IEEE International Conference on Software Maintenance and Evolution, 2015, pp. 281–290.

[169] N. Prollochs, S. Feuerriegel, D. Neumann, Detecting negation scopes for financial news sentiment using reinforcement learning, in: Proceedings of the Hawaii International Conference on System Sciences, 2016, pp. 1164–1173.

[170] A. Asmi, T. Ishaya, Negation identification and calculation in sentiment analysis, in: Proceedings of the Second International Conference on Advances in Information Mining and Management, 2012, pp. 1–7.

[171] R. Morante, A. Liekens, W. Daelemans, Learning the scope of negation in biomedical texts, in: Proceedings of the Conference on Empirical Methods in Natural Language Processing, 2008, pp. 715–724.

[172] L. Jia, C. Yu, , W. Meng, The effect of negation on sentiment analysis and retrieval effectiveness, in: Proceedings of the ACM Conference on Information and Knowledge Management, 2009, pp. 1827–1830.

[173] Y. Choi, C. Cardie, Learning with compositional semantics as structural inference for subsentential sentiment analysis, in: Proceedings of the Conference on Empirical Methods in Natural Language Processing, 2008, pp. 793–801.

[174] A. Kennedy, D. Inkpen, Sentiment classification of movie and product reviews using contextual valence shifters, Computational Intelligence 22 (2) (2006) 110–125.

[175] S. Wu, T. Miller, J. Masanz, M. Coarr, S. Halgrim, D. Carrell, C. Clark, Negation's not solved: Generalizability versus optimizability in clinical natura, PLoS ONE 9 (11) (2014) e112774.

[176] A. Bacchelli, A. Cleve, M. Lanza, A. Mocci, Extracting structured data from natural language documents with island parsing, in: Proceeding of the International Conference on Automated Software Engineering, 2011, pp. 476–479.

[177] A. Reyes, P. Rosso, D. Buscaldi, From humor recognition to irony detection: The figurative language of social media, Data and Knowledge Engineering 74 (2012) 1–12.

[178] A. Balahur, J. Hermida, A. Montoyo, Detecting implicit expressions of sentiment in text based on commonsense knowledge, in: Proceedings of the Workshop on Computational Approaches to Subjectivity and Sentiment Analysis, 2011, pp. 53–60.

[179] AlchemyLanguage: Natural language processing for advanced text analysis, http://www.alchemyapi.com/products/alchemylanguage/sentiment-analysis, last access: March 2019.

[180] E. Guzman, Visualizing emotions in software development projects, in: Proceedings of the Conference on Software Visualization, 2013, pp. 1–4.

[181] M. Ortu, G. Destefanis, S. Counsell, S. Swift, R. Tonelli, M. Marchesi, Arsonists or firefighters? Affectiveness in agile software development, in: Proceedings of the International Conference on Extreme Programming, 2016, pp. 144–155.

[182] E. Guzman, W. Maalej, How do users like this feature? A fine grained sentiment analysis of app reviews, in: Proceedings of the International Requirements Engineering Conference, 2014, pp. 153 – 162.

[183] M. Choudhury, M. Gamon, S. Counts, Happy, nervous or surprised? Classification of human affective states in social media, in: Proceedings of the International AAAI Conference on Weblogs and Social Media, 2012, pp. 435–438.

[184] A. Muhammad, N. Wiratunga, R. Lothian, R. Glassey, Domain-based lexicon enhancement for sentiment analysis, in: Proceedings of the SGAI International Conference on Artificial Intelligence, 2013.

[185] A. Murgia, M. Ortu, P. Tourani, B. Adams, An exploratory qualitative and quantitative analysis of emotions in issue report comments of open source systems, Empirical Software Engineering (2017) 1—44.

[186] Stack Exchange Data Dump, https://archive.org/details/stackexchange, last access: March 2019.

[187] J. Ding, H. Sun, X. Wang, X. Liu, Entity-level sentiment analysis of issue comments, in: Proceeding of the Third International Workshop on Emotion Awareness in Software Engineering, 2018.

[188] M. Islam, M. Zibran, A comparison of software engineering domain specific sentiment analysis tools, in: IEEE International Conference on Software Analysis, Evolution and Reengineering, 2018, pp. 487–491.

[189] N. Novielli, D. Girardi, F. Lanubile, A benchmark study on sentiment analysis for software engineering research, in: Proceedings of the International Conference on Mining Software Repositories, 2018, pp. 799–808.

[190] A. Abbasi, A. Hassan, M. Dhar, Benchmarking twitter sentiment analysis tools, in: Proceed-

ings of the International Conference on Language Resources and Evaluation, 2014, pp. 823–829.

[191] F. Ribeiro, M. Araújo, P. Gonçalves, M. Gonçalves, G. Benevenuto, SentiBench - A benchmark comparison of state-of-the-practice sentiment analysis methods, EPJ Data Science 5 (23) (2016) Open access.

[192] P. Gonçalves, M. Araújo, F. Benevenuto, M. Cha, Comparing and combining sentiment analysis methods, in: Proceedings of the First ACM Conference on Online Social Networks, 2013, pp. 27–38.

[193] S. Loria, Textblob: Simplified text processing, Secondary TextBlob: Simplified Text Processing, 2014.

[194] N. Pappas, G. Katsimpras, E. Stamatatos, Distinguishing the popularity between topics: A system for up-to-date opinion retrieval and mining in the web, in: Proceedings of the 14th International Conference on Computational Linguistics and Intelligent Text Processing, 2013, pp. 197–209.

[195] V. Narayanan, I. Arora, A. Bhatia, Fast and accurate sentiment classification using an enhanced naive bayes model, in: Proceedings of the 14th International Conference on Intelligent Data Engineering and Automated Learning, 2013, pp. 194–201.

[196] URL for downloading DEVA and Benchmark Dataset, https://figshare.com/s/277026f0686f7685b79e, verified: Dec 2017.

[197] C. Yang, K. Lin, H. Chen, Building emotion lexicon from weblog corpora, in: Annual Conference of the Association for Computational Linguistics, 2007, pp. 133–136.

[198] C. Yang, K. Lin, H. Chen, Writer meets reader: Emotion analysis of social media from both the writer's and reader's perspectives, in: Proceedings of the IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology, 2009, pp. 287–290.

[199] T. Allen, Bulletin boards of the 21st century are coming of age, Smithsonian 19 (6) (1988) 83–93.

[200] Z. Chuang, C. Wu, Emotion recognition from textual input using an emotional semantic network, in: Proceedings of the International Conference on Spoken Language Processing, 2002.

[201] List of interjections, https://www.vidarholen.net/contents/interjections/, verified: Aug 2017.

[202] M. Mäntylä, K. Petersen, T. Lehtinen, C. Lassenius, Time pressure: A controlled experiment of test case development and requirements review, in: Proceedings of the International Conference on Software Engineering, 2014, pp. 83–94.

[203] N. Nan, D. Harter, Impact of budget and schedule pressure on software development cycle time and effort, IEEE Transactions on Software Engineering 35 (5) (2009) 624–637.

[204] G. Miller, Wordnet: A lexical database for english, Communications of the ACM 38 (11) (1995) 39–41.

[205] A. Bacchelli, M. Lanza, R. Robbes, Linking e-mails and source code artifacts, in: Proceedings

of the International Conference on Software Engineering, 2010, pp. 375–384.

[206] W. Medhat, A. Hassan, H. Korashy, Sentiment analysis algorithms and applications: A survey, Ain Shams Eng 5 (4) (2014) 1093–1113.

[207] A. Yadollahi, A. Shahraki, O. Zaiane, Current state of text sentiment analysis from opinion to emotion mining, ACM Computing Surveys 50 (2) (2017) 1–33.

[208] G. Uddin, F. Khomh, Automatic summarization of api reviews, in: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, 2017, pp. 159–170.

[209] G. Uddin, F. Khomh, Opiner: An opinion search and summarization engine for apis, in: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, 2017, pp. 978–983.

[210] A. D'Andrea, F. Ferri, P. Grifoni, T. Guzzo, Approaches, tools and applications for sentiment analysis implementation, International Journal of Computer Applications 125 (3) (2015) 26–33.

[211] L. Polanyi, A. Zaenen, Contextual valence shifters, Computing Attitude and Affect in Text: Theory and Applications 20 (2006) 1–10.

[212] B. M. A. R. R. C. J. García, R. Alcaraz, Application of entropy-based metrics to identify emotional distress from electroencephalographic recordings, Entropy 18 (6).

[213] T. Eerola, J. Vuoskoski, A comparison of the discrete and dimensional models of emotion in music, Psychology of Music 39 (1) (2010) 18–49.

[214] Scikit-learn: Machine Learning in Python, http://scikit-learn.org/stable/, Verified: Sept 2018.

[215] M. Thelwall, Heart and soul: Sentiment strength detection in the social web with sentistrength,, in: CyberEmotions, 2013, pp. 1–14.

[216] D. Ye, Z. Xing, C. Foo, Z. Ang, J. Li, N. Kapre, Software-specific named entity recognition in software engineering social content, in: SANER, 2016, pp. 90–101.

[217] SyntaxNet: Neural Models of Syntax, https://github.com/tensorflow/models/tree/master/research/syntaxnet, Verified: Aug 2018.

[218] Industrial-Strength Natural Language Processing, https://spacy.io, Verified: Sept 2018.

[219] F. Omran, C. Treude, Choosing an nlp lib. for analyzing softw. documentation: A systematic lit. review and a series of exp., in: Proceedings of the International Conference on Mining Software Repositories, 2017, pp. 187–197.

[220] Snowball, http://snowballstem.org, Verified: Sept 2018.

[221] TF-IDF, http://www.tfidf.com, Verified: Aug 2018.

[222] L. Khreisat, Arabic text classification using n-gram frequency statistics a comparative study, in: CDM, 2006, pp. 78–82.

[223] The Social Media Glossary: 226 Essential Definitions, https://blog.hootsuite.com/social-media-glossary-definitions/, Verified: Aug 2018.

[224] M. Raymer, W. Punch, E. Goodman, L. Kuhn, A. Jain, Dimensionality reduction using genetic algo., IEEE Trans. on Evol. Comp. 4 (2) (2000) 164–171.

[225] W. Maalej, H. Happel, From work to word: How do software developers describe their work?, in: Proceedings of the International Conference on Mining Software Repositories, 2009, pp. 121–130.

[226] Y. Ayalew, K. Mguniin, An assessment of changeability of open source software, Computer and Information Science 6 (3) (2013) 68–79.

[227] G. Pinto, F. Castor, Y. Liu, Mining questions about software energy consumption, in: Proceedings of the International Conference on Mining Software Repositories, 2014, pp. 22–31.

[228] H. Malik, P. Zhao, M. Godfrey, Going green: An exploratory analysis of energy-related questions, in: Proceedings of the International Conference on Mining Software Repositories, 2015, pp. 418–421.

[229] I. Moura, G. Pinto, F. Ebert, F. Castor, Mining energy-aware commits, in: Proceedings of the International Conference on Mining Software Repositories, 2015, pp. 56–67.

[230] Z. Zeng, G. Roisman, T. Huang, A survey of affect recognition methods: audio, visual, and spontaneous expressions, IEEE Transactions on Pattern Analysis and Machine Intelligence 31 (1) (2009) 39–58.

[231] G. Yang, S. Baek, J. Lee, B. Lee, Analyzing emotion words to predict severity of softw. bugs: A case study of open source proj., in: Proceedings of the The ACM/SIGAPP Symposium On Applied Computing, 2017, pp. 1280–1287.

[232] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, P. Devanbu, On the “naturalness" of buggy code, in: Proceedings of the International Conference on Software Engineering, 2016, pp. 428–439.

[233] M. Islam, M. Zibran, A. Nagpal, Security vulnerabilities in categories of clones and non-cloned code: An empirical study, in: ESEM, 2017, pp. 20 – 29.

[234] D. Anderson, D. Sweeney, T. Williams, Statistics for Business and Economics, 10th Edition, Thomson Higher Education, 2009.

[235] T. Miller, S. Pedell, A. Lopez-Lorca, A. Mendoza, L. Sterling, A. Keirnan, Emotionled modelling for people-oriented requirements engineering: the case study of emergency systems, Journal of Systems and Software 105 (C) (2015) 54–71.

[236] R. Souza, C. Chavez, R. Bittencourt, Patch rejection in firefox: negative reviews, backouts, and issue reopening, Journal of Software Engineering Research and Development 3 (9) (2015) NA.

[237] J. Cheruvelil, B. Silva, Developers’ sentiment and issue reopening, in: IEEE/ACM 4th International Workshop on Emotion Awareness in Software Engineering, 2019, pp. 29–33.

[238] Q. Umer, H. Liu, Y. Sultan, Emotion based automated priority prediction for bug reports, IEEE Access 6 (2018) 35743–35752.

[239] Q. Umer, H. Liu, Y. Sultan, Sentiment based approval prediction for enhancement reports, The Journal of Systems and Software (57–69).

[240] G. Yang, T. Zhang, B. Lee, An emotion similarity based severity prediction of software bugs: A case study of open source projects, IEICE TRANS INF & SYST E101 (8) (2018) 2015–2016.

[241] G. Williams, A. Mahmoud, Modeling user concerns in the app store: A case study on the rise and fall of yik yak, in: 64-75 (Ed.), Proceedings of the International Requirements Engineering Conference, 2018.

[242] L. Carreno, K. Winbladh, Analysis of user comments: An approach for software requirements evolution, in: Proceedings of the International Conference on Software Engineering, 2013, pp. 582–591.

[243] Y. Jo, A. Oh, Aspect and sentiment unification model for online review analysis, in: Proceedings of the fourth ACM international conference on Web search and data mining, 2011, pp. 815–824.

[244] O. Shmueli, N. Pliskin, L. Fink, Explaining over-requirement in software development projects: An experimental investigation of behavioral effects, International Journal of Project Management 33 (2) (2014) 380–394.

[245] D. Kahneman, J. Knetsch, R. Thaler, Experimental tests of the endowment effect and the coase theorem, Journal of Political Economy 98 (6) (1990) 1325–1348.

[246] M. Norton, D. Mochon, D. Arielyc, The ikea effect: When labor leads to love, Journal of Consumer Psychology 22 (3) (2012) 453–460.

[247] N. Franke, M. Schreier, Why customers value self-designed products: The importance of process effort and enjoyment, Journal of Product Innovation Management 7 (2010) 1020–1031.

[248] B. Fu, J. Lin, L. Li, C. Faloutsos, J. Hong, N. Sadeh, Why people hate your app - making sense of user feedback in a mobile app store, in: Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2013, pp. 1276–1284.

[249] D. Blei, J. Lafferty, Dynamic topic models, in: Proceedings of the 23rd international conference on Machine learning, 2006, pp. 113–120.

[250] W. Jiang, H. Ruan, L. Zhang, P. Lew, J. Jiang, For user-driven software evolution: Requirements elicitation derived from mining online reviews, in: Proceedingd of the Pacific-Asia Conference on Knowledge Discovery and Data Mining, 2014, pp. 584–595.

[251] L. Zhao, A. Zhao, Sentiment analysis based requirement evolution prediction, Future Internet (2019) 1–14.

[252] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural Computation 9 (8) (1997) 1735–1780.

[253] E. Biswas, K. Shanker, L. Pollock, Exploring word embedding techniques to improve senti-

ment analysis of software engineering texts, in: IEEE/ACM 16th International Conference on Mining Software Repositories, 2019, pp. 68 – 78.

[254] F. Zhou, J. Jiao, X. Yang, B. Lei, Augmenting feature model through customer preference mining by hybrid sentiment analysis, Expert Systems with Applications 89 (2017) 306–317.

[255] L. Northrop, Sei's software product line tenets, IEEE Software 19 (4) (2002) 32–40.

[256] S. Panichella, A. Sorbo, E. Guzman, Ardoc: App reviews development oriented classifier, in: Proceedings of the International Symposium on the Foundations of Software Engineering, 2016, pp. 1023–1027.

[257] A. Sorbo, S. Panichella, C. Alexandru, J. Shimagaki, C. Visaggio, G. Canfora, H. Gall, What would users change in my app? summarizing app reviews for recommending software changes, in: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2016, pp. 499–510.

[258] A. Sorbo, S. Panichella, C. Alexandru, C. Visaggio, G. Canfora, SURF: summarizer of user reviews feedback, in: Proceedings of the IEEE/ACM International Conference on Software Engineering, 2017, pp. 55–58.

[259] F. Palomba, P. Salza, A. Ciurumelea, S. Panichella, Recommending and localizing change requests for mobile apps based on user reviews, in: Proceedings of the International Conference on Software Engineering, 2017, pp. 106–117.

[260] X. Gu, S. Kim, What parts of your apps are loved by users, in: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, Vol. 760–770, 2015.

[261] W. Maalej, H. Nabil, Bug report, feature request, or simply praise? On automatically classifying app reviews, in: Proceedings of the 25th International Requirements Engineering Conference, 2015, pp. 116–125.

[262] G. Williams, A. Mahmoud, Mining twitter feeds for software user requirements, in: Proceedings of the IEEE 25th International Requirements Engineering Conference, 2017, pp. 1–10.

[263] E. Guzman, M. El-Halaby, B. Bruegge, Ensemble methods for app review classification: An approach for software evolution, in: IEEE/ACM International Conference on Automated Software Engineering, 2015, pp. 771–776.

[264] E. Guzman, M. Ibrahim, M. Glinz, A little bird told me: Mining tweets for requirements and software evolution, in: Proceedings of the International Requirements Engineering Conference, 2017, pp. 11–20.

[265] N. Jha, A. Mahmoud, Mining non-functional requirements from app store reviews, Empirical Software Engineering 24 (2017) 3659–3695.

[266] M. Nayebi, H. Farrahi, G. Ruhe, Which version should be released to the app store?, in: ESEM, 2017, pp. 324–333.

[267] G. Uddin, F. Khomh, Automatic mining of opinions expressed about apis in stack overflow, IEEE Transaction on Software Engineering.

[268] B. Lin, F. Zampetti, G. Bavota, M. Penta, M. Lanza, Pattern-based mining of opinions in q&a websites, in: Proceedings of the International Conference on Software Engineering, 2019, pp. 548–559.

[269] L. Bradley, Measuring emotion: the self-assessment semantic differential., Journal of Behavior Therapy and Experimental Psychiatry 25 (1) (1994) 49–59.

[270] D. Graziotin, X. Wang, P. Abrahamsson, Happy software developers solve problems better: psychological measurements in empirical software engineering., PeerJ 2 (1).

[271] M. Ortu, G. Destefanis, S. Counsell, M. Marchesi, R. Tonelli, Connecting the dots: Measuring effectiveness and affectiveness in software systems, in: IEEE/ACM 2nd International Workshop on Emotion Awareness in Software Engineering, 2017, pp. 52–53.

[272] M. Ahasanuzzaman, M. Asaduzzaman, C. Roy, K. Schneider, Classifying stack overflow posts on api issues, in: Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering, 2018, pp. 244–254.

[273] C. Sutton, A. McCallum, An introduction to conditional random fields, Found. Trends Mach. Learn. 4 (4) (2012) 267–373.

[274] L. Breiman, Random forests, Machine Learning 45 (1) (2001) 5–32.

[275] M. Scott, A cluster analysis method for grouping means in the analysis of variance, Biometrics 30 (3) (1974) 507–512.

[276] A. Mondal, M. Rahman, C. Roy, Embedded emotion-based classification of stack overflow questions towards the question quality prediction, in: Proceedings of the International Conference on Software Engineering and Knowledge Engineering, 2016.

[277] S. Robertson, Understanding inverse document frequency: On theoretical arguments for idf, Journal of Documentation 60.

[278] K. Hornik, M. Stinchcombe, H. White., Multilayer feedforward networks are universal approximators, Neural Network (1989) 359–366.

[279] C. Cortes, V. Vapnik, Support-vector networks, Machine Learning (273–297) 1995.

[280] R. Serva, Z. Senzer, L. Pollock, K. Vijay-Shanker, Automatically mining negative code examples from software developer q & a forums, in: IEEE/ACM International Conference on Automated Software Engineering Workshop, 2015, pp. 115–122.

[281] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, J. Noble, Qualitas corpus: A curated collection of java code for empirical studies, in: Proceedings of the Asia Pacific Software Engineering Conference, 2010, pp. 336 – 345.

[282] R. Duda, P. Hart, D. Stork, Pattern Classification, A Wiley-Interscience Publication, 2000.

[283] L. Aversano, L. Cerulo, M. Di Penta, How clones are maintained: An empirical study, in: Proceedings of the European Conference on Software Maintenance and Reengineering, 2007, pp. 81–90.

[284] M. Zibran, C. Roy, A constraint programming approach to conflict-aware optimal schedul-

ing of prioritized code clone refactoring, in: Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation, 2011, pp. 105–114.

[285] H. Sajnani, V. Saini, C., A comparative study of bug patterns in java cloned and non-cloned code, in: Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation, 2014, pp. 21–30.

[286] C. Roy, M. Zibran, R. Koschke, The vision of software clone management: Past, present, and future, in: CSMR-18/WCRE-21 Software Evolution Week (SEW'14), 2014, pp. 18–33.

[287] R. Al-Ekram, C. Kapser, R. Holt, M. Godfrey, Cloning by accident: An empirical study of source code cloning across software systems, in: Proceedings of the International Symposium on Empirical Software Engineering, 2005, pp. 376–385.

[288] M. Gabel, Z. Su, A study of the uniqueness of source code, in: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, pp. 147–156.

[289] R. Saha, M. Asaduzzaman, M. Zibran, C. Roy, K. Schneider, Evaluating code clone genealogies at release level: An empirical study, in: Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation, 2010, pp. 87–96.

[290] M. Zibran, R. Saha, C. Roy, K. Schneider, Evaluating the conventional wisdom in clone removal: A genealogy-based empirical study, in: Proceedings of the The ACM/SIGAPP Symposium On Applied Computing, 2013, pp. 1123–1130.

[291] M. Zibran, R. Saha, C. Roy, K. Schneider, Genealogical insights into the facts and fictions of clone removal, ACM Applied Computing Review 13 (4) (2013) 30–42.

[292] H. Brar, P. Kaur, Comparing detection ratio of three static analysis tools, Int. Journal of Computer Applications 124 (13) (2015) 35–40.

[293] RATS - Rough Auditing Tool for Security, http://www.securesw.com/rats/, July 2017.

[294] J. Viega, J. Bloch, T. Kohno, G. Macgraw, Token-based scanning of source code for security problems, ACM Transactions on Information and System Security 5 (3) (2002) 238–261.

[295] R. McLean, Comparing static security analysis tools using open source software, in: SSRC, 2012, pp. 68–74.

[296] J. Wilander, M. Kamkar, A comparison of publicly available tools for dynamic buffer overflow prevention, in: Proceedings of the 10th Network and Distributed System Security Symposium, 2003, pp. 124–138.

[297] D. Pozza, R. Sisto, L. Durante, A. Valenzano, Comparing lexical analysis tools for buffer overflow detection in network software, in: Proceedings of the International Conference on Communication System Software and Middleware, 2006, pp. 126–133.

[298] G. Tan, J. Croft, An empirical security study of the native code in the jdk, in: SS, 2008, pp. 365–377.

[299] A. Sotirov, Automatic vulnerability detection using static source code analysis, Master's the-

sis, The University of Alabama (2005).

[300] NIST Software Assurance Reference Dataset, https://samate.nist.gov/SRD/, July 2017.

[301] J. Islam, M. Mondal, C. Roy, Bug replication in code clones: An empirical study, in: Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering, 2016.

[302] M. Rahman, C. Roy, On the relationships between stability and bug-proneness of code clones: An empirical study, in: SCAM, 2017, pp. 131–140.

[303] M. Mondal, C. K. Roy, K. A. Schneider, Identifying code clones having high possibilities of containing bugs, in: Proceedings of the International Conference on Program Comprehension, 2017.

[304] M. Zibran, C. Roy, Conflict-aware optimal scheduling of code clone refactoring: A constraint programming approach, in: Proceedings of the International Conference on Program Comprehension, 2011, pp. 266 – 269.

[305] N. Nagappan, T. Ball, A. Zeller, Mining metrics to predict component failures, in: 2006, Proceedings of the International Conference on Software Engineering, pp. 452–461.

[306] V. Saini, H. Sajnani, C. Lopes, Comparing quality metrics for cloned and non-cloned java methods: A large scale empirical study, in: Proceedings of The International Conference on Software Maintenance and Evolution, 2016, pp. 256–266.

[307] JGit, https://www.eclipse.org/jgit/, verified: Jan 2018.

[308] J. Islam, M. Mondal, C. Roy, K. Schneider, A comparative study of software bugs in clone and non-clone code, in: Proceedings of the International Conference on Software Engineering and Knowledge Engineering, 2017, pp. 436–443.

[309] R. Snelick, A. Mink, M. Indovina, A. Jain, Large-scale evaluation of multimodal biometric authentication using state-of-the-art systems, IEEE Trans. on Pattern Analysis and Machine Intelligence 27 (3) (2005) 450–455.

[310] J. Svajlenko, C. K. Roy, Evaluating modern clone detection tools, in: ICSME, 2014, pp. 321 – 330.

[311] G. Greene, B. Fischer, Cvexplorer: Identifying candidate developers by mining and exploring their open source contributions, in: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, 2016, pp. 804–809.

[312] H. Osman, M. Lungu, O. Nierstrasz, Mining frequent bug-fix code changes, in: Proceedings of the IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, 2014, pp. 343–347.

[313] X. Wang, Y. Dang, L. Zhang, D. Zhang, E. Lan, H. Mei, Can i clone this piece of code here?, in: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, 2012, pp. 170–179.

[314] M. Mondal, C. Roy, M. Rahman, R. Saha, J. Krinke, K. Schneider, Comparative stability of

cloned and non-cloned code: An empirical study, in: Proceedings of the The ACM/SIGAPP Symposium On Applied Computing, 2012, pp. 1227–1234.

[315] M. Mondal, C. Roy, K. Schneider, Dispersion of changes in cloned and non-cloned code, in: IWSC, 2012, pp. 29–35.

[316] M. Kim, D. Notkin, Discovering and representing systematic code changes, in: Proceedings of the International Conference on Software Engineering, 2009, pp. 309–319.

[317] S. Kim, K. Pan, E. Whitehead, Memories of bug fixes, in: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2006, pp. 35–45.

[318] K. Pan, S. Kim, E. W. Jr., Toward an understanding of bug fix patterns, Empirical Software Engineering 14 (3) (2009) 286–315.

[319] J. Falleri, F. Morandat, X. Blanc, M. Martinez, M. Monperrus, Fine-grained and accurate source code differencing, in: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, 2014, pp. 313–324.

[320] K. Huang, B. Chen, X. Peng, D. Zhou, Y. Wang, Y. Liu, W. Zhao, Cldiff: Generating concise linked code differences, in: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, 2018, pp. 679–690.

[321] J. Maletic, M. Collard, Supporting source code difference analysis, in: Proceedings of the IEEE International Conference on Software Maintenance, 2004, p. PP.

[322] B. Fluri, M. Wursch, M. Pinzger, H. Gall, Change distilling: Tree differencing for fine-grained source code change extraction, IEEE Transactions on Software Engineering 33 (11) (2007) 725–743.

[323] M. Hashimoto, A. Mori, Diff/TS: A tool for fine-grained structural change analysis, in: Proceedings of the Working Conference on Reverse Engineering, 2008, pp. 279–288.

[324] D. Kim, J. Nam, J. Song, S. Kim, Automatic patch generation learned from human-written patches, in: Proceedings of the International Conference on Software Engineering, 2013, pp. 802–811.

[325] M. Martinez, L. Duchien, M. Monperrus, Automatically extracting instances of code change patterns with ast analysis, in: Proceedings of the IEEE International Conference on Software Maintenance, 2013, pp. 22 – 28.

[326] M. Martinez, M. Monperrus, Mining software repair models for reasoning on the search space of automated program fixing, Empirical Software Engineering 20 (1) (2015) 176–205.

[327] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, M. Maia, Dissection of a bug dataset: Anatomy of 395 patches from Defects4J, in: Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering, 2018.

[328] R. Yue, N. Meng, Q. Wang, A characterization study of repeated bug fixes, in: Proceedings of the International Conference on Software Maintenance and Evolution, 2017, pp. 422–432.

[329] R. Saha, Y. Lyu, H. Yoshida, M. Prasad, Elixir: Effective object-oriented program repair, in:

Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, 2017, pp. 648–659.

[330] P. Fournier-Viger, C. Wu, A. Gomariz, V. Tseng, VMSP: Efficient vertical mining of maximal sequential patterns, Advances in Artificial Intelligence 8436 (2014) 83–94.

[331] J. Wang, J. Han, C. Li, Frequent closed sequence mining without candidate maintenance, IEEE Trans. on Knowledge Data Engineering 19 (8) (2007) 1–15.

[332] P. Fournier-Viger, J. Lin, A. Gomariz, T. Gueniche, A. Soltani, Z. Deng, The SPMF open-source data mining library version 2, in: Proceedings of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases, 2016, pp. 36–40.

[333] I. Miliaraki, K. Berberich, R. Gemulla, S. Zoupanos, Mind the gap: Large-scale frequent sequence mining, in: Proceedings of the International Conference on Management of Data, 2013, pp. 797–808.

[334] X. Jin, J. Han, K-Medoids Clustering, Encyclopedia of Machine Learning (Springer), 2011.

[335] A. jain, Data clustering: 50 years beyond k-means, Pattern Recognition Letters 31 (8).

[336] R. Tibshirani, G. Walther, T. Hastie, Estimating the number of clusters in a data set via the gap statistic, Journal of the Royal Statistical Society 63 (2) (2001) 411–423.

[337] Python-String-Similarity, https://github.com/luozhouyang/python-string-similarity/blob/master/README.md, last access: Jan 2019.

[338] PyCluster - Clustering module for Python, https://bioconda.github.io/recipes/pycluster/README.html, last access: Jan 2019.

[339] A. Cantor, Sample-size calculations for cohen's kapp, Psychological Methods 1 (2) (1996) 150–153.

[340] A. Mockus, L. Votta, Identifying reasons for software changes using historic databases, in: Proceedings of the International Conference on Software Maintenance and Evolution, 2000, pp. 120–130.

[341] J. Bevan, E. Whitehead, S. Kim, M. Godfrey, Facilitating software evolution research with kenyon, in: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, pp. 177–186.

[342] T. Nguyen, H. Nguyen, N. Pham, J. Al-Kofahi, T. Nguyen, Graph-based mining of multiple object usage patterns, in: Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering, 2009, pp. 383–392.

[343] R. Just, D. Jalali, M. Ernst, Defects4j: A database of existing faults to enable controlled testing studies for java programs, in: Proceedings of the International Symposium on Software Testing and Analysis, 2014, pp. 437–440.

[344] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, D. Poshyvanyk, On learning meaningful code changes via neural machine translation, in: Proceedings of the International Conference on Software Engineering, 2019, p. (to appear).

[345] GumtreeSpoon - Spoon version of GumTree, https://github.com/SpoonLabs/gumtree-spoon-ast-diff, last access: Jan 2019.

[346] M. Soto, F. Thung, C. Wong, C. Goues, D. Lo, A deeper look into bug fixes: Patterns, replacements, deletions, and additions, in: Proceedings of the International Conference of Mining Software Repositories, 2016, pp. 512–515.

[347] Q. Hanam, F. Brito, A. Mesbah, Discovering bug patterns in javascript, in: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2016, pp. 144–156.

[348] S. Sudhakrishnan, J. Madhavan, E. W. Jr., Understanding bug fix patterns in verilog, in: Proceedings of the International Working Conference on Mining Software Repositories, 2015, pp. 39–42.

[349] F. Long, M. Rinard, Automatic patch generation by learning correct code, in: Proceedings of the Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2016, pp. 298–312.

[350] H. Oumarou, N. Anquetil, A. Etien, S. Ducasse, K. Taiwe, Identifying the exact fixing actions of static rule violation, in: Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering, 2015, pp. 371–379.

[351] R. Rolim, G. Soares, R. GheyI, T. Barik, L. D'Antoni, Learning quick fixes from code repositories, in: arXiv preprint arXiv:1803.03806, 2018.

[352] B. Cole, D. Hakim, D. Hovemeyer, R. Lazarus, W. Pugh, K. Stephens, Improving your software using static analysis to find bugs, in: Proceedings of the ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications, 2006, pp. 673–674.

[353] B. Fluri, E. Giger, H. Gall, Discovering patterns of change types, in: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, 2008, pp. 15–19.

[354] T. Molderez, R. Stevens, C. Roover, Mining change histories for unknown systematic edits, in: Proceedings of the International Conference on Mining Software Repositories, 2017, pp. 248–256.

[355] M. Kim, J. Beall, D. Notkin, Discovering and representing logical structure in code change., Tech. rep., University of Washington (2007).

[356] M. Kim, D. Notkin, D. Grossman, Automatic inference of structural changes for matching across program versions., in: Proceedings of the International Conference on Software Engineering, 2007.

[357] S. Breu, T. Zimmermann, Mining aspects from version history, in: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, 2006, pp. 221–230.

[358] S. Negara, M. Codoban, D. Dig, R. Johnson, Mining fine-grained code changes to detect unknown change patterns, in: Proceedings of the International Conference on Software En-

gineering, 2014, pp. 803–813.

[359] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, L. Zhang, Precise condition synthesis for program repair, in: Proceedings of the International Conference on Software Engineering, 2017, pp. 416–426.

[360] S. Gievska, K. Koroveshovski, T. Chavdarova, A hybrid approach for emotion detection in support of affective interaction, in: CDMW, 2014, pp. 352–359.

[361] N. Imtiaz, J. Middleton, P. Girouard, E. Murphy-Hill, Sentiment and politeness analysis tools on developer discussions are unreliable, but so are people, in: Proceedings of the 3rd International Workshop on Emotion Awareness in Software EngineeringJune, 2018, pp. 55–61.

[362] B. Lin, F. Zampetti, R. Oliveto, M. Penta, M. Lanza, G. Bavota, Two datasets for sentiment analysis in software engineering, in: IEEE International Conference on Software Maintenance and Evolution (ICSME), 2018, pp. 712–712.

Appendix A

Publications Out of This Dissertation Research

Parts of this thesis work have been published in journals, conferences, and workshops of Software Engineering, as well as under preparation for submission. The list of the publications is mentioned below.

a) Refereed Journal Contributions

[**a1**] **Md Rakibul Islam** and Minhaz F. Zibran. SentiStrength-SE: Exploiting Domain Specificity for Improved Sentiment Analysis in Software Engineering Text. Elsevier Journal of Systems and Software (JSS), 145: 125-146, 2018.

[**a2**] **Md Rakibul Islam**, Minhaz F. Zibran. Exploration and Exploitation of Developers' Sentimental Variations in Software Engineering. International Journal of Software Innovation, 4 (4): 35 - 55, 2016.

b) Refereed Conference & Workshop Contributions

[**b3**] **Md Rakibul Islam** and Minhaz F. Zibran. How Bugs Are Fixed: Exposing Bug-fix Patterns with Edits and Nesting Levels. In proceedings of the 35th ACM/SIGAPP Symposium on Applied Computing (SAC), pp. NA, Brno, Czech Republic, 2020 (forthcominmag)

[**b4**] **Md Rakibul Islam**, Md Kauser Ahmmed and Minhaz F. Zibran. MarValous: machine learning based detection of emotions in the valence-arousal space in software engineering text. In Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing (SAC), pp. 1786-1793, Limassol, Cyprus, 2019.

[**b5**] **Md Rakibul Islam** and Minhaz F. Zibran. DEVA: Sensing Emotions in the Valence Arousal Space in Software Engineering Text. In Proceedings of the 33rd ACM/SIGAPP Symposium On Applied Computing (SAC), pp. 1536 - 1543, France, 2018.

[**b6**] **Md Rakibul Islam** and Minhaz F. Zibran. A Comparison of Software Engineering Domain Specific Sentiment Analysis Tools. In Proceedings of the 25th IEEE International

Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 487 - 491, Italy, 2018.

[b7] **Md Rakibul Islam** and Minhaz F. Zibran. Sentiment Analysis of Software Bug Related Commit Messages. In Proceedings of the 27th International Conference on Software Engineering and Data Engineering (SEDE), USA, 2018 (**Winner of Best Paper Award**).

[b8] **Md Rakibul Islam**, Minhaz F. Zibran, and Aayush Nagpal. Security Vulnerabilities in Categories of Clones and Non-Cloned Code: An Empirical Study. In Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 20-29, Canada, 2017.

[b9] **Md Rakibul Islam** and Minhaz F. Zibran. A Comparison of Dictionary Building Methods for Sentiment Analysis in Software Engineering Text. In Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 478 - 479, Canada, 2017.

[b10] **Md Rakibul Islam** and Minhaz F. Zibran. Leveraging Automated Sentiment Analysis in Software Engineering. In Proceedings of the 14th International Conference on Mining Software Repositories (MSR), pp.203 - 214, Buenos Aires, Argentina, May 2017.

[b11] **Md Rakibul Islam** and Minhaz F. Zibran. Towards Understanding and Exploiting Developers' Emotional Variations in Software Engineering. In Proceedings of the 14th IEEE International Conference on Software Engineering Research, Management and Applications (SERA), pp. 185-192, Baltimore, Maryland, USA, 2016 (**invited at the International Journal of Software Innovation**).

[b12] **Md Rakibul Islam** and Minhaz F. Zibran. On the Characteristics of Buggy Code Clones: A Code Quality Perspective. In Proceedings of the 12th IEEE International Workshop on Software Clones (IWSC), pp. 23 - 29, Italy, 2018.

[b13] **Md Rakibul Islam** and Minhaz F. Zibran. A Comparative Study on Vulnerabilities in Categories of Clones and Non-Cloned Code. In Proceedings of the 10th IEEE International Workshop on Software Clones (IWSC), pp. 8 - 14, Osaka, Japan, 2016 (**Winner of Best Paper Award**).

c) Poster Presentation

[c14] **Md Rakibul Islam** and Minhaz F. Zibran. Entity Based Aspect-Oriented Opinion Mining in Software Engineering. InnovateUNO, New Orleans, LA, USA, 2019.

[c15] **Md Rakibul Islam** and Minhaz F. Zibran. An Empirical Study of Security Vulnerabilities in Software Systems. In 10th EAI International Conference on Digital Forensics & Cyber Crime (ICDF2C 2018), New Orleans, USA, 2018.

[c16] **Md Rakibul Islam** and Minhaz F. Zibran. Understanding Bug Fix Patterns: Towards an Improved Automated Program Repair Method. InnovateUNO, New Orleans, LA, USA, 2018.

d) Miscellaneous

[d17] **Md Rakibul Islam** and Minhaz F. Zibran. Insights into Continuous Integration Build Failures. In Proceedings of the 14th International Conference on Mining Software Repositories (MSR), pp. 467-470, Buenos Aires, Argentina, May 2017.

[d18] Duaa Alwad, Manisha Panta, Minhaz F. Zibran and **Md Rakibul Islam**. An Empirical Study of the Relationships between Code Readability and Software Complexity, In Proceedings of the 27th International Conference on Software Engineering and Data Engineering (SEDE), USA, 2018.

[d19] Naw Safrin Sattar, Md A. M. Faysal, Minhaz Zibran, Shaikh Arifuzzaman, and **Md Rakibul Islam**. Data Mining in-IDE Activities: Why Software Developers Fail, In Proceedings of the 27th International Conference on Software Engineering and Data Engineering (SEDE), USA, 2018.

The publications [**a1**] and [**b10**] constitute the Chapter 4. The publications [**b5**] and [**b4**] contribute in writing Chapter 6 and Chapter 7, respectively. Chapter 5 is composed of the performance comparison of the existing sentiment analysis tools that appears in the publication [**b6**]. Chapter 8 corresponds to the publications [**a2**] and [**b11**]. The publication [**b7**] contributes in writing Chapter 9. The empirical studies appeared in [**b8**], [**b12**], and [**b13**] constitute the Chapter 12, Chapter 13, and Chapter 11, respectively. The empirical study of bug-fixing patterns appeared in the publication [**b3**] constitutes the Chapter 14. The publications [**c14**], [**c15**], and [**c16**] are also related to this thesis as a whole.

A.1 Co-authorship

In this thesis I have presented my own research conducted under the supervision of Minhaz F. zibran. Citations are properly mentioned for the ideas and techniques that are not products of my own work. In cases where citations are not available, we describe the ideas and techniques in such a way that indicates they existed prior to this work.

Small parts of the research presented in this thesis involved joint work with other researchers to make co-authored publications. Md Kauser Ahmmed and Aayush Nagpal are two co-authors of the publications [**b4**] and [**b8**], respectively. Md Kauser Ahmmed was the lead developer in transforming my ideas into a machine learning based emotions mining tool `MarValous`. Aayush Nagpal helped in conducting data analysis to present in the publication [**b8**]. I am the sole contributor in initial formulation of the idea, data collection, writing, and shaping the results of the published research papers [**b4]**, [**b8]**.

Other than the aforementioned contributions of the co-authors of my papers, the entire work presented in this thesis is the outcome of my own research carried out under the supervision of Minhaz F. zibran.

Vita

Md Rakibul Islam has successfully defended his PhD thesis in Computer Science department at The University of New Orleans (UNO), Louisiana, USA. After hafing his degree, in Fall 2020, he will join as a tenure-track Assistant Professor in the Computer Science Department at the University of Wisconsin- Eau Claire. His research interests include: (i) Human Aspects in Software Engineering, (ii) Software Security, (iii) Source Code Analysis, and (iv) Natural Language Processing. He often blends his research interests and apply various techniques, such as Machine Learning, Data Mining, and Graph Theories to come up with useful tools and interesting insights (achieved empirically) for greater benefits of concerned communities. He has co-authored more than 15 papers in different journal and venues that include MSR, SANER, ESEM, SAC and others.