A NOVEL BUG TRIAGING STRATEGY USING DEVELOPER RECOMMENDATION AND LOAD BALANCING MODEL

K. M. Aslam Uddin
Registration Number: 100
Session: 2018-2019

A Thesis
Submitted in Partial Fulfillment of the Requirements for the Degree of
Master of Philosophy (MPhil)

MASTER OF PHILOSOPHY (MPHIL)

Institute of Information Technology
University of Dhaka
Dhaka, Bangladesh

© K. M. Aslam Uddin, 2024

Dhaka University Institutional Repository

A NOVEL BUG TRIAGING STRATEGY USING DEVELOPER RECOMMENDATION AND LOAD BALANCING MODEL

K. M. Aslam Uddin

Approved:

Signature Date

_______________________________________ ____________

Supervisor: Dr. Kazi Muheymin-Us-Sakib

Signature Date

_______________________________________ ____________

Co-supervisor: Dr. Ahmedul Kabir

Dhaka University Institutional Repository

Abstract

Bug triage is essential in efficiently assigning bugs to developers by leveraging past experiences. Without this crucial process, experienced developers may be inundated with assignments, while newer developers may be underutilized. Furthermore, improper bug distribution among different developer types can lead to various issues, including delays, errors, decreased capacity, and diminished job satisfaction. Previous bug triaging methods often do not account for newly joined developers, making them ineffective in recommending these developers for bug assignments. Consequently, these methods lead to improper task allocation, denying new team members valuable learning opportunities during bug resolution. Furthermore, prior research tends to overlook workload distribution among different developer categories, neglecting the need to balance bug assignments among experienced developers, newcomers, and those with varying skill levels. To address these issues, there is a need for an automated bug triaging technique that not only includes new developers but also prioritizes workload distribution among different developer categories. Therefore, this study introduces a novel bug triaging strategy that combines two pivotal models: Bug Solving Developer Recommendation Model (BSDRM) and Developer Scheduler (DevSched).

The first model, known as the BSDRM, forms the core of automated bug triaging. BSDRM harnesses the power of Machine Learning (ML) algorithms and historical bug reports to intelligently suggest developers for specific bug resolution tasks. To achieve this, Eclipse, Mozilla, and NetBeans datasets are aggregated and split into training and testing sets. Subsequently, a sentence-embedded model is crafted from the training set, generating a developer-specific word repository. In contrast, the test set is transformed into a vocabulary list using an embedded model. BSDRM identifies eligible developers by matching their developer-specific word repository with the bug report vocabulary list via K-Nearest Neighbour (KNN) analysis. These developers are then categorized into three groups: experienced, newly experienced, and fresh graduate developers, utilizing a classification model comprising various ML algorithms Decision Tree (DT), Extra Tree (ET), AdaBoost (AdC), Bagging Classifier (BC), Gradient Boosting (GB), KNN, Nearest Centroid (NC), Bernoulli Naïve Bayes (BNB), Multinomial Naïve Bayes (MNB), Complement Naïve

Dhaka University Institutional Repository

Bayes (CoNB), Gaussian Na¨ıve Bayes (GNB), Logistic Regression (LR), Perceptron (Pr), and Multi-Layer Perceptron (MLP). Remarkably, the Bagging Classifier exhibits outstanding performance, achieving 96.59% accuracy in classifying developers with varying experience levels.

In tandem with BSDRM, this study introduces the second model, DevSched, which assumes a critical role in balancing developer workloads. DevSched factors in workload distribution, developer proficiency, and bug characteristics. It generates multiple developer profiles based on historical bug reports and assigns bugs to developers by assessing the highest similarity between bug vectors and developer corpora. DevSched also dynamically adjusts developer workloads and refines their ratings based on performance. The study utilizes bug reports from Eclipse, Mozilla, and NetBeans to evaluate developer performance in the bug-triaging process. DevSched efficiently assigns and balances bugs among various developer categories, resulting in significantly reduced standard deviations for Eclipse, NetBeans, and Mozilla datasets compared to conventional bug distribution processes. This meticulous process is reiterated for each bug, ensuring optimal resource allocation and timely resolution of critical issues.

The proposed study will collectively enhance bug resolution efficiency, optimize developer workloads, and ensure that both experienced and newer developers are judiciously utilized in the bug triaging process.

Acknowledgements

I begin by expressing my profound gratitude to Allah, the Most Merciful and Most Compassionate, for guiding me throughout this academic journey and granting me the strength and wisdom to undertake this research.

I am indebted to my supervisor, Dr. Kazi Muheymin-Us-Sakib, whose unwavering support, valuable insights, and expert guidance have been instrumental in shaping this thesis. Your mentorship and dedication to my academic growth have left an indelible mark on my journey.

I extend my sincere appreciation to my co-supervisor, Dr. Ahmedul Kabir, for your valuable input, constructive feedback, and encouragement throughout this research. Your expertise has greatly enriched my work.

I would like to acknowledge the support and encouragement of my colleagues and friends who stood by me during the ups and downs of this academic endeavor. Your camaraderie has been a source of strength and inspiration.

I am grateful to my family for their unwavering support, love, and understanding. Your sacrifices and encouragement have been my driving force.

Last but not least, I would like to express my gratitude to the faculty and staff of Institute of Information Technology, University of Dhaka for providing the resources and friendly environment necessary for conducting this research.

Thank you, everyone, for being a part of this journey and for contributing to the successful completion of my MPhil degree thesis.

List of Publications

1. Uddin, K.A., Kowsher, M. and Sakib, K., 2022, September. BSDRM: A Machine Learning Based Bug Triaging Model to Recommend Developer Team. In International Conference on Machine Intelligence and Emerging Technologies (pp. 256-270). Cham: Springer Nature Switzerland.
2. Uddin, K.A., Satu, M.S., Riyad, M.M.H. and Sakib, K., 2023. DevSched: an efficient bug-triaging model for allocating and balancing developer tasks. Iran Journal of Computer Science, pp.1-11.

Contents

Approval ii

Abstract iii

Acknowledgements v

List of Publications vi

Table of Contents vii

List of Tables xi

List of Figures xii

1 Introduction 1

1.1 Motivation………………………………………………………………………… 2
1.2 Research Question………………………………………………………………… 5
1.3 Contribution of the Research…………………………………………………… 8
1.4 Scope of the Research…………………………………………………………… 9
1.4 Structure of the Thesis………………………………………………………….. 9

2 Background Study 11

2.1 Software Bug……………………………………………………………………… 12
2.1.1 Causes of Software Bugs……………………………………………………… 12
2.1.2 Life Cycle of Software Bug…………………………………………………... 14
2.1.3 Stages of Software Bug……………………………………………………….. 17
2.1.4 Classification of Software Bug……………………………………………….. 19
2.1.4.1 Severity of Bugs…………………………………………………….. 19
2.1.4.2 Nature of the Bug…………………………………………………… 21
2.1.4.3 Platform……………………………………………………………... 23
2.1.4.4 Version……………………………………………………………… 24
2.1.4.5 Components………………………………………………………… . 25
2.1.4.6 Metadata Tagging…………………………………………………… 27
2.1.5 Software Bugs Handling Process……………………………………………... 29
2.2 Bug Reports……………………………………………………………………… . 33
2.2.1 Bug Features Description……………………………………………………... 33
2.2.2 Bug Report History…………………………………………………………… 35
2.3 Bug Triaging for Developer Recommendation………………………………… 38
2.3.1 Bug Submission……………………………………………………………….. 38
2.3.2 Data Collection and Pre-processing…………………………………………... 40
2.3.2.1 Stemming…………………………………………………………… . 41
2.3.2.2 Stop Word Removal………………………………………………… 43
2.3.3 Indexing……………………………………………………………………….. 43

2.3.4 Graph Construction………………………………………………………… 46
2.3.5 Categorization……………………………………………………………. 47
2.3.6 Prioritization……………………………………………………………… 47
2.3.7 Assignment and Notification…………………………………………….. 48
2.3.8 Update Developer and Bug Profile………………………………………. 49
2.4 Load Balancing in Bug Triggering………………………………………………... 49
2.5 Summary…………………………………………………………………………… 52

3 Literature Review **53**
3.1 Existing Research Work………………………………………………………… 54
3.2 Topic Model Based Approach…………………………………………………... 54
3.2.1 DRETOM………………………………………………………………… 54
3.2.2 BUTTER………………………………………………………………….. 55
3.2.3 Developer Ranking Algorithm……………………………………………. 56
3.2.4 BAHA…………………………………………………………………….. 57
3.3 Information Retrieval Based Approach…………………………………………. 58
3.3.1 Mining Software Repositories……………………………………………. 58
3.3.2 Leveraging Latent Semantic Indexing……………………………………. 59
3.3.3 Modeling Developer Expertise…………………………………………… 59
3.3.4 Enhanced LDA Methods…………………………………………………. 60
3.3.5 Entropy-Based Optimization……………………………………………... 60
3.3.6 Expertise Scoring and Ranking…………………………………………… 60
3.4 Social Network Analysis-Based Approach……………………………………... 61
3.4.1 Social Network Analysis-Based Approach……………………………….. 61
3.4.2 Social Network Analysis with Machine Learning…………………………... 61
3.4.3 Information Retrieval for Bug Similarity………………………………… 61
3.4.4 Bug-Fixing Expertise and Association-Based……………………………. 62
3.4.5 Concept Profile and Social Network……………………………………... 62
3.5 Dependency Based Approach…………………………………………………… 64
3.5.1 Bug Dependency-Based Mathematical Model…………………………… 64
3.5.2 Scheduling-Driven Task Assignment…………………………………….. 64
3.5.3 Automated Bug Triage with Dependency………………………………... 64
3.5.4 Dependency-Aware with NLP and Integer Programming…………………... 65
3.6 Machine Learning-Based Bug Triage Systems…………………………………. 65
3.6.1 Conventional Machine Learning-Based…………………………………….. 66
3.6.1.1 Bug Triage Using Selected Fields……………………………….. 66
3.6.1.2 Fuzzy Set Features for Bug Triage………………………………. 66
3.6.1.3 Generalized Recommendations for Developers…………………. 66
3.6.1.4 Developer Prioritization with TF-IDF…………………………… 66
3.6.1.5 Bug Triage with Metadata Consideration……………………….. 67
3.6.1.6 Automatic Developer Assignment with Discriminatory Terms…. 67
3.6.1.7 Data Reduction for Improved Accuracy…………………………. 67
3.6.1.8 Ensemble Classifier for Enhanced Bug Triage………………….. 67
3.6.1.9 SVM-Based Bug Recommender System………………………… 67
3.6.1.10 Incorporating Categorical Features and Metadata………………. 68
3.6.1.11 High-Confidence Bug Triage……………………………………. 68
3.6.1.12 Semi-Automated with Skill-Based Developer…………………... 68

3.6.1.13 Enhanced Bug Triage through Integrated Models………………. 69
3.6.1.14 Activity-Based Bug Triage Strategy…………………………….. 69
3.6.1.15 BugFixer………………………………………………………. 70
3.6.1.16 Search-Based Bug Triage with Apache Lucene…………………. 71
3.6.1.17 Common Vocabulary-Based Bug Triage Algorithm…………….. 71
3.6.2 Deep Learning-Based…………………………………………………….. 72
3.6.2.1 CNN-Based Bug Triage with Word2Vec……………………….. 72
3.6.2.2 Multilabel Deep Neural Network for Bug Triage……………….. 72
3.6.2.3 Deep Bidirectional RNN with Attention for Bug Triage………... 72
3.6.2.4 Activity-Based Bug Triage with CNN…………………………... 72
3.6.2.5 CNN-Based Bug Fixer Recommendation System……………….. 73
3.6.2.6 Heterogeneous Graph-Based Bug Triage………………………... 73
3.6.2.7 Multitriage Model for Developer Assignment………………….. 73
3.6.2.8 Bug Triage with Graph Neural Network………………………… 73
3.7 Reassignment-Based Approaches………………………………………………… 74
3.7.1 Tossing Graph-Based Approaches……………………………………….. 74
2.7.1.1 Bug Tossing Graphs……………………………………………... 74
2.7.1.2 Enhanced Tossing Graphs………………………………………. 75
3.7.2 Multi-Feature Incremental Learning……………………………………….. 75
3.8 Text Categorization Based Approaches…………………………………………… 76
3.8.1 Bug Report Meta Data…………………………………………………… 76
3.8.2 Developer Preference Elicitation…………………………………………. 76
3.8.3 Code Authorship………………………………………………………….. 77
3.9 Bug Data Reduction Approaches…………………………………………………. 78
3.9.1 Source-Based Bug Assignment Approaches……………………………... 78
3.9.2 Developer Vocabulary-Based Approach…………………………………. 79
3.9.3 Commit Time-Based Approach………………………………………….. 79
3.10 Cost Aware Based Approaches…………………………………………………… 80
3.11 Industry-Oriented Approaches……………………………………………………. 81
3.11.1 Research-Industry Cooperation………………………………………….. 81
3.11.2 Team Assignment……………………………………………………….. 81
3.12 Summary………………………………………………………………………… 82

4 Recommend Developer Team Efficiently **84**

4.1 Overview of BSDRM……………………………………………………………… 85
4.1.1 Dataset Description……………………………………………………….. 85
4.1.1.1 Eclipse…………………………………………………………. 87
4.1.1.2 Mozilla………………………………………………………… 88
4.1.1.3 NetBeans……………………………………………………… 88
4.1.2 Generating Developer Matrix…………………………………………….. 88
4.1.2.1 ED Profile……………………………………………………... 88
4.1.2.2 NED Profile…………………………………………………… 89
4.1.2.3 FG Profile…………………………………………………….. 89
4.1.3 Training Stage……………………………………………………………. 90
4.1.3.1 Sentence Embedding………………………………………….. 90
4.1.3.2 Balancing Data………………………………………………… 91
4.1.3.3 Employing Different classifiers……………………………….. 92

4.1.4 New Task………………………………………………………………. 102
4.1.5 Exploring Eligible Developer…………………………………………….. 102
4.1.6 Classifying Developers…………………………………………………… 103
4.1.6.1 Evaluation Metrics…………………………………………….. 103
4.1.7 Forming Developer Team…………………………………………………… 106
4.2 Result Analysis……………………………………………………………………….. 106
4.2.1 Classification Results of BSDRM………………………………………... 107
4.2.2 Comparisons with Existing Works……………………………………….. 109
4.3 Summary………………………………………………………………………………… 111

5 Task Allocation and Load Balancing **112**
5.1 Overview of DevSched……………………………………………………………….. 113
5.1.1 Data Pre-processing………………………………………………………. 114
5.1.2 Developer Profile Rating Calculation ..…………………………………….. 114
5.1.3 Load Creation…………………………………………………………….. 115
5.1.3.1 Data Transformation…………………………………………... 115
5.1.3.2 Bug Distribution………………………………………………. 116
5.1.4 Load Balancing…………………………………………………………… 116
5.1.5 Update Developer Profile………………………………………………… 119
5.2 Experimental Setting……………………………………………………………… 120
5.2.1 Data Description………………………………………………………….. 120
5.2.2 Environment Setup……………………………………………………….. 120
5.3 Result Analysis………………………………………………………………………. 121
5.3.1 Results of Eclipse Dataset………………………………………………… 122
5.3.2 Results of Mozilla Dataset………………………………………………... 124
5.3.3 Results of NetBeans Dataset…………………………………………….. 126
5.3.4 Comparative Analysis among Datasets…………………………………... 127
5.3.5 Comparisons with Existing Works……………………………………….. 128
5.4 Summary…………………………………………………………………………………. 129

6 Discussion and Conclusion **131**
6.1 Summary of Results……………………………………………………………….. 132
6.1.1 Recommend Developer Team Efficiently…………………………………… 132
6.1.2 Task Allocation and Load Balancing……………………………………... 134
6.2 Impact on Software Companies…………………………………………………… 135
6.3 Limitations of the Study……………………………………………………………. 137
6.4 Future Research…………………………………………………………………… 137

Bibliography **139**

List of Tables

2.1	Causes of software bugs…………………………………………………………….	13
2.2	Classification of bugs based on severity………………………………………………	20
2.3	Classification of bugs based on the nature of the bug………………………………..	22
2.4	Bug classification based on platform………………………………………………….	24
2.5	Bug classification based on version…………………………………………………..	25
2.6	Bug classification based on components……………………………………………..	26
2.7	Bug categorization based on metadata tagging………………………………………..	28
2.8	Example of stemming……………………………………………………………….	41
2.9	Algorithms of stemming…………………………………………………………….	42
2.10	Different indexing process………………………………………………………….	44
4.1	Dataset description…………………………………………………………………..	87
4.2	Developer classification result………………………………………………………	108
4.3	Comparison of BSDRM with traditional models……………………………………...	110
5.1	Date ranging…………………………………………………………………………	120
5.2	Load thresholds of different types of developer……………………………………….	122
5.3	The distribution of developer by implementing DevSched……………………………	124
5.4	Comparative Analysis in terms of standard deviation…………………………………	128
5.5	Comparisons with existing works	129

List of Figures

2.1	Life cycle of a bug [23]………………………………………………………………..	15
2.2	Stages of bug [24]…………………………………………………………………...	18
2.3	Overview of software bug classification………………………………………………	20
2.4	Software bug handling process………………………………………………………..	31
2.5	Automated bug assignment process [27]……………………………………………..	32
2.6	Bug tracking system of TFS [27]……………………………………………………	32
2.7	Sample bug reporting form of Bugzilla [28]…………………………………………..	34
2.8	Bug report history [29]……………………………………………………………...	36
2.9	Bug history creation process……………………………………………………….	37
2.10	Bug triaging process for developer recommendation…………………………………	39
2.11	Sample Developer-Component-Bug network [46]……………………………………	46
2.12	Load balancing in bug triaging………………………………………………………..	50
3.1	Overview of BUTTER model [48]……………………………………………………	56
3.2	Overall framework of Developer Ranking Algorithm [49]……………………………	57
3.3	Overall framework of BAHA [50]…………………………………………………….	58
3.4	Bug concept with the topic terms, where θ=0.05 [65]…………………………………	63
3.5	An example of social network [65]……………………………………………………	63
3.6	Structure of hybrid bug triaging [55]………………………………………………….	70
3.7	Overall structure of BugFixer……………………………………………………….	71
4.1	The workflow diagram of BSDRM…………………………………………………..	86
5.1	DevSched: Task allocation and load balancing model……………………………….	113
5.2	The standard deviation curve for Eclipse dataset……………………………………..	123
5.3	The standard deviation curve for Mozilla dataset…………………………………….	125
5.4	The standard deviation curve for NetBeans dataset…………………………………..	126

Chapter 1

Introduction

In the world of software development, dealing with software bugs is a big challenge. These bugs can range from small annoyances to causing major system failures [1]. They pose a constant threat to both software application reliability and user satisfaction. The onus of promptly and effectively resolving these bugs falls squarely upon software development teams, with the efficiency of this process rooted in the intricate practice of bug triaging. This complex procedure determines how bugs are efficiently assigned to developer teams or individuals based on factors such as severity, frequency, and risk [2]. It is a crucial part of the software development process, helping decide which bugs are most important and how to use resources to fix them.

When a new bug is discovered, it usually involves many skilled developers to solve it quickly. However, this can lead to an uneven distribution of bug-fixing tasks. Experienced developers may end up with lots of bugs to fix, while newer and mid-level developers struggle to find opportunities to learn and help. Sometimes, developers with expertise in different areas are brought in to fix bugs, which make it even harder to balance the workload and decide who should work on what. To make sure tasks are distributed fairly and that developers can share their knowledge and experience, we need a system that smartly assigns bugs to different types of developers based on their skills and bug-solving expertise.

Numerous methods have been proposed in previous works to gauge the impact of bug triaging [3-5]. However, these past reports often lack any record of newly joined developers, rendering these techniques ineffective in including them in the final bug assignment recommendations. Consequently, existing methods often result in improper task allocation among developers, leaving new team members without opportunities to gain and share knowledge through the bug resolution process. Furthermore, it is worth noting that prior

research has typically overlooked the crucial aspect of workload distribution among various categories of developers. This omission is a significant limitation as it neglects the need to balance the assignment of bugs among experienced developers, new joiners, and developers with varying skill levels. In light of these issues, there is a pressing need for an automatic bug triaging technique that not only considers the inclusion of new developers but also prioritizes workload distribution among different developer categories. Such an approach can contribute to more effective bug resolution processes, fostering knowledge sharing and skill development among all team members, regardless of their experience level.

This study presents a novel bug-triaging strategy using developer recommendation and load balancing model where an ML-based bug-triaging approach known as the Bug Solving Developer Recommendation Model is proposed to tackle the challenge of assigning efficient developers to bug resolution processes. Additionally, the study introduces the Developer Scheduler, a task allocation and load balancing model, designed to optimize the distribution of unassigned bugs among developers of varying expertise levels.

In this chapter, the motivation for this thesis is briefly presented, accompanied by an introduction to the research questions. Subsequently, the contribution of this research is succinctly discussed. Finally, the scope and organization of the thesis are outlined.

1.1 Motivation

Due to the growing complexity of software systems, an inevitable consequence is the emergence of a substantial number of bugs in software projects. Statistical reports indicate that addressing these bugs consumes a significant portion, approximately 45%, of the total development time for software companies [6]. In response to this challenge, bug tracking systems, such as Bugzilla [7] and Jira [8], have assumed an increasingly pivotal role in the management of software bugs. When a bug report is submitted to the bug tracking system, the behind-the-scenes management necessitates the manual assignment of the bug report to the most suitable developer based on its description. This process, known as bug triaging, is a labor-intensive and time-consuming endeavor, further complicated by two primary challenges [9]. Firstly, the sheer volume of bug reports can be staggering, particularly in the case of sizable projects, where bug tracking systems routinely receive a deluge of bug reports daily. For instance, the Eclipse project alone encounters approximately 91 bug reports every day [6]. Secondly, the multitude of developers involved in bug resolution poses another obstacle. Backstage managers often find it impractical to be intimately acquainted with the skill levels

of all developers, leading to manual bug triaging that may not effectively allocate bug reports to the most suitable individuals. In light of these challenges, the adoption of automatic bug report triaging methods has become imperative within the realm of software testing. These methods offer a solution to the aforementioned problems, streamlining the bug management process and enhancing its efficiency.

In addressing the challenge of bug triaging, one must acknowledge its inherent complexities. Bug reports, pivotal in resolving software issues, encompass diverse types, including usability concerns, intricate functionality issues, and critical security matters. Effective triaging requires a deep understanding of each report's unique characteristics and the expertise required for resolution. The bug's severity also plays a crucial role, ranging from minor inconveniences to critical system failures, demanding precise allocation to skilled developers [10]. Bug complexity adds to the challenge, with some requiring extensive domain knowledge and others a nuanced understanding. Developer availability, workloads, commitments, and expertise must be considered. Matching developers' skills with the right bugs and understanding their preferences ensures efficient resolution. Navigating this complexity is vital for timely and effective bug resolution, enhancing software quality and the user experience.

Automated bug triaging techniques rely on developers' bug-fixing profiles, usually derived from their source code commits or bug-fixing histories [11]. Upon receiving a new bug report, a thorough process unfolds, wherein the report's keywords are meticulously examined in relation to the source code and previous bug reports. Developers with the closest match to the report are considered potential candidates to resolve the bug [12]. However, these methods have limitations, notably the lack of consideration for collaboration among developers. This means they might miss opportunities for teamwork in tackling complex issues. There are also issues like suggesting inexperienced or inactive developers, struggling with changing project dynamics, and challenges in accommodating developers' varying expertise in different domains [13]. These limitations highlight the need for more comprehensive bug assignment solutions to address these challenges thoroughly.

To build effective developer teams in bug triaging, it's crucial to consider how developers collaborate. Many team-based assignment techniques rely on historical bug reports to assess developer expertise and communication [14, 15]. However, this approach has a limitation: it doesn't account for newly onboarded developers. This oversight can result in these developers being excluded from bug resolution tasks. It's essential to address this issue because new developers frequently join software development teams, each bringing

unique skills and knowledge. Integrating these newcomers enriches the collective knowledge base and promotes knowledge sharing and skill development, enhancing the efficiency of the software development cycle. New developers also bring fresh perspectives, innovative solutions, and renewed energy, which are valuable for adapting to evolving project requirements and industry trends. Embracing newcomers not only leverages fresh talent but also fosters a culture of continuous learning and growth, leading to more robust and innovative software solutions.

The significance of load balancing in the allocation of tasks among developers in bug triaging cannot be emphasized enough. Surprisingly, no previous studies have delved into this crucial aspect of bug management. Load balancing stands as a fundamental pillar in ensuring the efficiency, effectiveness, and overall health of the bug triaging process. It plays a pivotal role in allowing software development teams to distribute bug-fixing tasks equitably among developers. This balanced workload allocation maximizes the efficient utilization of available resources, thus avoiding overburdening some developers and underutilizing others, which, in turn, fosters a more productive and harmonious work environment. Additionally, it serves as a proactive measure to mitigate the risk of bottlenecks within the bug triaging process. Without proper load distribution, there is a real danger of some developers becoming overwhelmed by an excessive number of tasks. This, in turn, may lead to delays in bug resolution and potentially disrupt the overall project timeline. Balanced task allocation effectively minimizes these bottlenecks, ensuring that bug reports are addressed promptly and that the project's progress remains consistent. It's crucial to acknowledge that different bugs may necessitate varying levels of expertise and domain knowledge for resolution. Load balancing comes to the rescue by facilitating the assignment of bugs to developers with the most appropriate skill sets. This optimization of expertise utilization results in quicker and more effective bug resolution. Furthermore, a well-balanced workload doesn't merely enhance individual efficiency; it fosters collaboration among developers. When team members are not overwhelmed by their individual tasks, they have more opportunities to share insights, discuss solutions, and collaborate on particularly challenging bug reports. This collaborative synergy often leads to the development of more innovative and robust resolutions. Another compelling aspect is the impact of task allocation on the well-being of developers. Overloading them with excessive work can lead to burnout and reduced morale. Conversely, balanced task allocation helps maintain a healthy work-life balance, preventing developer fatigue and preserving their motivation and enthusiasm. A content and motivated team is more likely to produce high-quality results. Recognizing the importance of load

balancing in task allocation within bug triaging not only optimizes efficiency but also safeguards the well-being of the development team, resulting in a win-win situation for both developers and the software development process.

Motivated by the ever-increasing complexities inherent in modern software development, significant research challenges emerge on the horizon of bug triaging. The imperative need for an automatic expert and recent team allocation technique that seamlessly integrates both existing and new developers remains a pressing concern. As software projects progress, incorporating new talent and efficiently utilizing experienced professionals becomes crucial for improving bug resolution processes. Furthermore, there is an interesting research gap that deserves our focus: no previous study has explored the use of load balancing techniques among different developer types in bug triaging. These challenges arise from real-world complexities and the needs of agile software development, emphasizing the necessity of addressing them thoroughly to optimize bug triaging methods. In doing so, we aim to not only elevate software quality but also streamline project efficiency, thereby contributing to the continued advancement of software engineering practices.

1.2 Research Question

The main objective of this research is to enhance the efficiency and effectiveness of bug triaging and developer assignment processes within the software development domain. This is accomplished through the development and application of innovative Machine Learning-based bug triaging models, the Bug Solving Developer Recommendation Model, aimed at recommending developers for bug resolution tasks in a balanced and efficient manner, considering factors like expertise, workload, and bug severity. Additionally, the research introduces the Developer Scheduler model to optimize bug distribution among developers, avoiding overloading experienced developers and ensuring opportunities for newer team members to gain experience. An essential goal is the inclusion of newly joined developers in the bug resolution process while balancing the workload among the developers, thereby enhancing software quality assurance within the software development pipeline. This leads to the following questions of this research:

R1: How can BSDRM enhance proper developer assignment in bug resolution processes?

In the dynamic landscape of software development, efficient bug triage and developer assignment are critical components for enhancing productivity and software quality assurance. To tackle the challenges of unbalanced developer assignments and diverse expertise levels, we introduce the BSDRM. This innovative ML-based approach is tailored to recommend developers for specific bug resolution tasks, considering their experience, workload, and expertise. This research question divides the inquiry into several sub-questions to comprehensively address the challenges. These sub-questions determine bug-solving preferences in different types of developers, classify developers, and recommend an accurate developer team.

a) **How does BSDRM determine the initial bug-solving preference of experienced and new developers?**

   BSDRM determines the initial bug-solving preference of experienced developers by creating developer profiles from existing bug reports. From the existing bug report, the sentence-embedded model generates the bag of developer's words. On the other hand, the new developers or fresh graduates do not have enough working experience in these circumstances. So, their histories are not available in the existing bug repository. The authority can give a predefined form to assess new developers' skills such as academic knowledge, technical expertise, interests, developed projects, etc.

b) How does BSDRM classify developers into different experience levels?

   BSDRM classifies developers into three categories: experienced, newly experienced, and fresh graduate developers. It achieves this through a developer classification model consisting of various classifiers like Decision Tree, Extra Tree, AdaBoost, Bagging Classifier, Gradient Boosting, KNN, Nearest Centroid, Bernoulli Naïve Bayes, Multinomial Naïve Bayes, Complement Naïve Bayes, Gaussian Naïve Bayes, Logistic Regression, Perceptron, and Multi-Layer Perceptron.

c) How does BSDRM recommend a developer team to solve testing/new bugs?

   On arrival of a new bug report, BSDRM applies the pre-trained sentence embedding model and extracts a vocabulary list for developers. An unsupervised KNN finder matches the vocabulary list of testing reports with the existing bag of developer's words. It identifies K's nearest developers who are eligible to resolve this bug. Using the developer classifier, identify developers with different experience levels, enabling the formation of balanced teams capable of efficiently addressing bug reports. Based

on the developer classification model results, BSDRM recommends a developer team to solve testing/new bugs.

***R2: How does the DevSched model optimize the distribution of unassigned bugs among developers with varying expertise levels and* ensure the balancing of developer tasks?**

The DevSched is introduced as a pivotal model in this research, designed to address the challenges posed by unassigned bugs and varying expertise levels among developers. DevSched plays a central role in optimizing bug distribution, ensuring equitable workload among different types of developers, and enhancing knowledge sharing. This research question unfolds into several sub-questions to comprehensively explore its role in optimizing bug distribution among developers with varying expertise and updating their profile ratings. These sub-questions investigate the determination of the effective allocation of bugs among new and experienced developers and the balancing of developer workload.

a) **How does DevSched rating different types of developer and update their profiles?**

DevSched updates different types of developer profiles based on the priority (Pt) and severity of the solved bugs. Generally, five types of priority (P1 −P5) and seven types of severity (Enhancement (Eh), Trivial (Tr), Minor (Mn), Normal (Nl), Major (MJ), Critical (Cr), Blocker (Bl)) used in the bug report. DevSched divides the seven types of severity into low (trivial, enhancement), medium (major, normal, minor), and high (blocker, critical) severity. DevSched calculate the rating using the following equations:

$$P_t = P_1 \times 5 + P_2 \times 4 + P_3 \times 3 + P_4 \times 2 + P5 \times 1 \quad \ldots\ldots\ldots\ldots\ldots..(1)$$

$$S_v = (E_h + T_r) \times 1 + (M_n + N_l + M_J) \times 2 + (C_r + B_l) \times 3 \quad \ldots\ldots\ldots(2)$$

From Eq. 1 and 2, $R_t = P_t + S_v$

After achieving a minimum number of points, the developer profile will be updated.

b) **How does DevSched ensure the balancing of the developer workload?**

DevSched ensures the balancing of the developer workload by using the load-balancing strategy. Initially, we calculate the developer's current workload and overall average threshold. New bugs are assigned to the developers based on their current workload and bug severity. If the developer's workload is less than the average threshold, a new task is given to the top-scoring developers to balance the current workload. Otherwise, the bug is assigned to the next top-scoring developer whose workload is less than the average threshold.

1.3 Contribution of this Research

The contributions of this study are encapsulated in two innovative models, each addressing crucial challenges within the software development domain. The first model, Bug Solving Developer Recommendation Model, harnesses the power of machine learning to revolutionize the bug triaging process. BSDRM achieves this by meticulously collecting and processing diverse datasets, enabling the creation of a robust developer recommendation framework. Through advanced techniques such as sentence embedding, vocabulary generation, and developer categorization, BSDRM offers precise bug-to-developer assignment, effectively optimizing bug triaging. This model empowers software teams by recommending developers of varying expertise levels, fostering an environment of efficient bug resolution.

In tandem, the second model, DevSched, focuses on the critical aspects of task allocation and load balancing in the bug triaging landscape. DevSched enhances developer profiles, transforms bug reports into vector representations, and dynamically adapts workloads based on bug resolution performance. By minimizing standard deviations in bug distributions, DevSched promotes workload balance, resource allocation, and bug resolution efficiency.

This study extends its reach to evaluate the practicality and effectiveness of these models across various software development projects, considering the adaptability and scalability factors. Real-world bug reports from prominent software projects serve as the testing ground to measure the impact of these models on developer performance and overall bug resolution efficiency. Furthermore, the research explores the implications of workload distribution, job satisfaction, and resource allocation in the software development landscape. By emphasizing the inclusion of new developers and the proper management of task assignments, this research aims to contribute significantly to the optimization of bug resolution processes, fostering a collaborative and efficient environment for developers, regardless of their experience level. In summary, this study provides valuable insights and solutions to the persistently challenging realm of bug triaging in software development, ultimately advancing the field and benefiting the software development industry as a whole.

1.4 Scope of the Research

This comprehensive research endeavors to address critical challenges within the domain of software development, specifically focusing on the intricate process of bug triaging. It utilizes a diverse dataset extracted from Eclipse, Mozilla, and NetBeans projects, spanning years of bug reports. The research delves into the creation of sentence-embedded models, generation of vocabulary lists, and developer classification to categorize developers into three distinct groups: experienced, newly experienced, and fresh graduates. This study is also designed to facilitate efficient task allocation and load balancing among developers, thereby reducing delays, minimizing errors, and enhancing job satisfaction. The proposed system collectively contributes to streamlining bug triaging processes, reducing delays, and enhancing job satisfaction among developers, ultimately propelling the software development industry toward higher efficiency and effectiveness.

1.5 Thesis Structure

This thesis is organized into several logically structured chapters, each serving a distinct purpose in presenting and analyzing the research findings on bug triaging and developer recommendation models. The following section outlines the contents of each chapter, providing a roadmap for readers to navigate through the study's comprehensive exploration of bug resolution efficiency and developer workload management.

Chapter 2: Background Study

In this chapter, the foundational concepts necessary for comprehending the proposed technique are elucidated. An in-depth exploration of software bugs, including their classification, is presented to establish a solid conceptual foundation. Furthermore, this chapter unveils the overarching architecture underpinning automated bug triaging for developer recommendation and load balancing, laying the groundwork for the subsequent chapters' detailed discussions and empirical findings.

Chapter 3: Literature Review

This chapter delves into a comprehensive examination of existing bug triaging techniques. It not only highlights the strengths and weaknesses of these methods but also elucidates the associated research gaps and unresolved issues, fostering a deeper comprehension of the current state of bug triaging practices.

Chapter 4: Recommend Developer Team Efficiently

This chapter is dedicated to a detailed exploration of the Bug Solving Developer Recommendation Model, a pivotal component of this research. BSDRM is the cornerstone of our approach, and this chapter delves into its inner workings, methodologies, and outcomes. It encompasses a comprehensive analysis of the model's functionality and performance in recommending developer teams efficiently for bug resolution tasks. We will explore the model's architecture, the datasets it utilizes, the machine learning techniques employed, and the evaluation metrics used to gauge its effectiveness. This chapter serves as a vital bridge between the theoretical framework and the practical application of our proposed bug triaging system.

Chapter 5: Task Allocation and Load Balancing

This chapter delves into the intricacies of the Developer Scheduler, a core component of our research focused on load balancing in bug triaging. DevSched plays a crucial role in optimizing the allocation of unassigned bugs among developers with varying expertise levels. This chapter provides an in-depth examination of DevSched's underlying principles, methodologies, and practical implementation. It encompasses an analysis of how DevSched creates developer profiles, assigns bugs to developers, and dynamically balances their workloads. We explore its impact on bug resolution efficiency and developer job satisfaction. This chapter serves as a bridge between theory and practical application, shedding light on the practical aspects of our bug triaging system's load balancing capabilities.

Chapter 6: Discussion and Conclusions

In this concluding chapter, we synthesize the key findings and insights garnered from our research endeavors. We engage in a comprehensive discussion, dissecting the implications of our BSDRM and DevSched in the realm of bug triaging. We evaluate the practical utility and effectiveness of these models in enhancing bug resolution processes. Additionally, we reflect on the contributions of this study to the field of software development and bug triaging methodologies. Furthermore, we explore avenues for future research and development, offering valuable suggestions for extending and refining the proposed techniques. This chapter culminates our research journey, underscoring the significance of our work in advancing bug triaging practices and paving the way for continued innovation in software development.

Chapter 2

Background Study

In the realm of software development, the critical practice of bug triage stands as an integral process where each reported bug is meticulously prioritized, considering factors such as severity, frequency, and risk, among others. This systematic classification is essential for rationalizing resource allocation and ensuring the efficient enhancement of software quality. However, the constant influx of bug reports from various software companies poses a significant challenge. When a new bug report surfaces, it often requires the expertise of several developers for a prompt resolution. Yet, this practice unintentionally burdens experienced developers with numerous bug assignments, leaving newer and mid-skilled developers struggling to find their place in bug-solving tasks. At times, experienced developers from unrelated domains are brought in to address bugs, further complicating the challenges of load balancing and developer allocation in bug triage. A system is needed to ensure a balanced distribution of tasks, promote knowledge sharing, and facilitate skill development among developers. This system should intelligently assign bugs to different categories of developers based on their bug-solving skills and expertise.

Efforts to address this problem have witnessed the emergence of several methodologies focusing on understanding the ramifications of bug assignments. Previous models have contributed significantly to the domain of bug triage [18-20]. However, these approaches often lacked the inclusion of new developers in their bug assignment recommendations, inadvertently perpetuating improper task allocation among developers. Bug Solving Developer Recommendation Model emerged as a solution to bridge the existing gaps. BSDRM leverages machine learning to recommend an adept developer team, comprising seasoned experts, medium-level fixers, and fresh graduates, for addressing newly arrived bugs. Merging multiple datasets (Eclipse, Mozilla, and NetBeans) and employing a

range of ML classifiers, BSDRM uses ML to suggest a proficient developer team, including experienced professionals, mid-level fixers, and fresh graduates, for handling newly reported bugs. By amalgamating these developer groups, BSDRM creates a bug-fixing dream team, ensuring that developers of varying experience levels receive opportunities to participate in the resolution of recently reported bugs. This collaborative approach lightens the load on experienced developers and imparts valuable bug-solving knowledge to newcomers. Meanwhile, to address the issue of workload distribution among different types of developers, the Developer Scheduler enters the fray. DevSched systematically allocates bugs based on predefined thresholds, competencies, and workloads, fostering a more balanced bug resolution process. In this chapter, we delve into the intricacies of these models, exploring their underlying mechanisms and evaluating their contributions to the realm of bug triage.

2.1 Software Bugs

Software bugs are inevitable and pervasive in the software development lifecycle [21]. These glitches, errors, or defects can manifest in various forms, ranging from minor annoyances to critical system failures. Understanding the nature of software bugs and the processes for handling them is crucial for maintaining the reliability and quality of software applications.

2.1.1 Causes of Software Bugs

Software will always have bugs. Software bugs can arise from mistakes and oversights while gathering and analyzing user prerequisites, preparing a program, assembling its source code, and interacting with hardware and software. Even if the code runs strictly as mandated by the distinct circumstances, we still need help if the planned outcome has to be more satisfactorily specified. Software bugs can arise from various factors and can be categorized into several common causes, as shown in Table 2.1. Identifying and addressing these multiple causes of software bugs requires a combination of best practices, code reviews, thorough testing, and adherence to coding standards. An effective software development process includes bug prevention and early detection strategies to minimize defects' impact on software quality and user satisfaction.

Table 2.1: Causes of software bugs

Cause	Description
Coding Errors	Programming mistakes made by developers are a primary source of software bugs. These errors can include syntax errors, logic errors, and incorrect use of programming constructs. Simple mistakes, such as typos or inaccurate variable assignments, can lead to unexpected behavior in the software.
Algorithmic Issues	Errors in the design or implementation of algorithms can result in software bugs. Algorithms are the fundamental building blocks of software, and mistakes in their logic can lead to incorrect results or unexpected behaviors.
Inadequate Testing	Only complete or sufficient testing can uncover existing bugs in the software. Test cases may only cover some possible scenarios or may not be executed rigorously enough to expose defects.
Integration Problems	Software is often composed of multiple components or modules that must interact seamlessly. Integration issues, such as incompatible interfaces, communication problems between modules, or data mismatches, can introduce bugs.
Environmental Variability	Software can behave differently in various environments, such as operating systems, hardware configurations, or network conditions. Bugs related to environmental factors can be challenging to reproduce and diagnose.
Concurrency and Multithreading	In multi-threaded or concurrent software, race conditions and synchronization issues can lead to bugs. These problems occur when multiple threads or processes access shared resources simultaneously, causing unpredictable behavior.
Memory Management	Memory-related bugs, such as memory leaks and buffer overflows, can lead to crashes, instability, or security vulnerabilities. These issues occur when software fails to allocate or release memory correctly.
Continue on next page	

Table 2.1 – continued from previous page

Cause	Description
Platform Dependencies	Bugs can arise due to dependencies on specific libraries, frameworks, or software versions. Changes in external dependencies can introduce compatibility issues.
External Factors	Software may interact with external services, APIs, or data sources. Bugs can occur when these external components change behavior or become unavailable.
Misunderstood Requirements	Bugs can result from a misunderstanding of user or system requirements. When developers misinterpret what the software is supposed to do, it can lead to functionality that does not meet expectations.
Unforeseen Interactions	Complex software systems may have many components that interact in intricate ways. Bugs can emerge when developers fail to anticipate or test all interactions between these components.
Legacy Code	Inheritances from legacy codebases can introduce bugs into modern software. Outdated or poorly documented code can be challenging to understand and modify without introducing defects.
Human Error	Human factors, including typos, oversight, and miscommunication among team members, can contribute to software bugs. Collaboration and communication are crucial in preventing such errors.
External Attacks	Security vulnerabilities often result from deliberate efforts to exploit software weaknesses. These vulnerabilities can lead to severe bugs when attackers gain unauthorized access or cause the software to behave maliciously.

2.1.2 Life Cycle of Software Bug

At the SLDC level, any software development team member can create an error. This error may be produced in the requirement, design, or coding phases [22]. If an error is made in the requirement phase and not solved in the same phase, an error is produced in the design phase.

Again, if an error is not solved in the design phase and passes in the coding phase, it creates a bug. This way, mistakes, and bugs attended and traveled at several phases of SDLC, as shown in Figure 2.1. The life cycle of a software bug can be classified into two stages.

(i). Bugs-in phase

(ii).Bugs-out phase

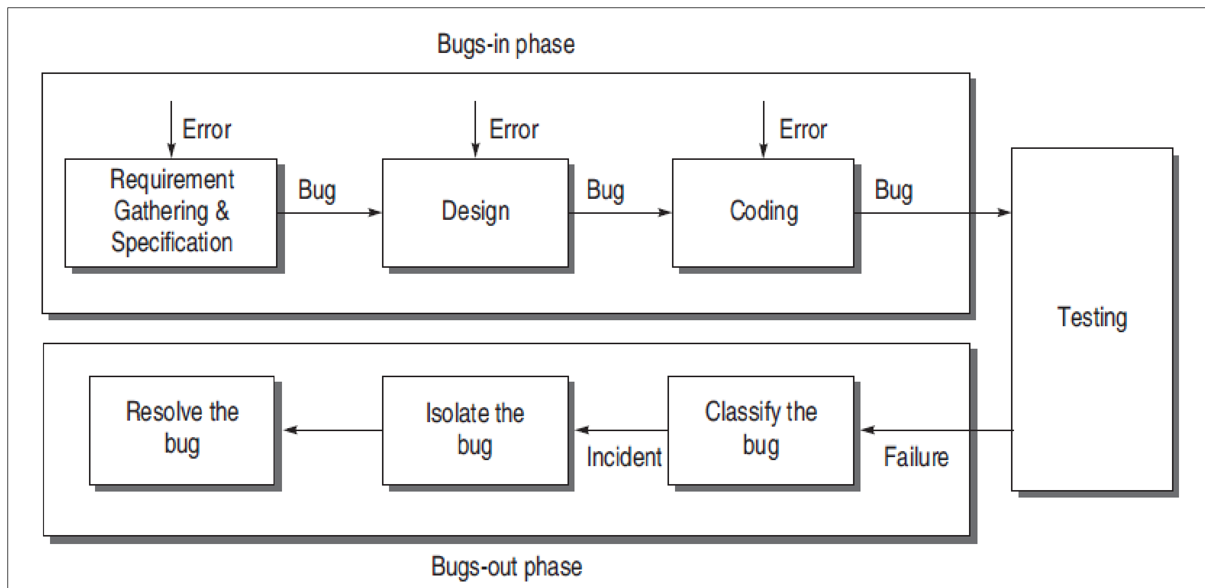


Figure 2.1: Life cycle of a bug [23]

i) Bugs-In Phase

This step introduces bugs in the software life cycle. When we make a mistake, it creates an error in a specific place of the software, and consequently, when this error goes unnoticed, it causes a bug in the software. These bugs are carried into the next stage of SDLC if not marked. If the verification is not performed earlier, these bugs cannot be identified. This phase encompasses several critical steps, each with its potential for introducing bugs.

a) Requirement Gathering and Specification

The first type of bug is present at the requirements collection and specification stage. Most bugs are present at this stage. Even though it is appropriately written, only some can be done if there are any requirements, no matter the software code. If these bugs are not detected, they can spread to later stages. It is challenging to convert the requirements collected from the end-user into the required specifications. The conditions may not be what the end user wants. Specification issues lead to wrong or missing features.

b) Design

In this stage, the software's architecture and high-level design are planned. This includes defining the system's structure, components, and interfaces. Design flaws or

misunderstandings can lead to bugs. If the design does not align with the specified requirements or if there are logical errors, they can manifest as bugs in the later stages.

c) Coding

This is where the actual implementation of the software takes place. Developers write the source code based on the design specifications. This stage is notorious for introducing bugs. Common coding-related issues include syntax errors, logic errors, and issues related to variables, loops, and conditional statements. Inadequate testing during coding can also result in undiscovered bugs.

ii) Bugs-Out Phase

The "bugs-out" phase of the software development life cycle (SDLC) involves the identification, classification, isolation, and resolution of bugs that were introduced during the "bugs-in" phase. This phase is essential for ensuring that the software is of high quality and free from defects. Here's an overview of the key steps in the "bugs-out" phase.

a) Bug Classification

Once a bug is identified, it must be classified or categorized based on various attributes, such as severity, priority, and type. Classification helps in prioritizing which bugs should be addressed first. Bugs are typically classified into different categories, such as:

Severity: Determining how critical the bug is to the system's functionality. Bugs are often classified as critical, major, minor, or trivial.

Priority: Establishing the order in which bugs should be fixed based on factors like business impact and customer requirements.

Type: Categorizing bugs based on their nature, such as functional, performance-related, security, or usability issues.

b) Bug Isolation

After classification, the next step is to isolate the bug, which involves identifying the specific area or component of the software where the bug resides. This step helps developers narrow down the source of the problem. Developers and testers work together to reproduce the bug in a controlled environment. This often involves replicating the conditions or actions that trigger the bug to understand its root cause. Debugging tools and logs may be used to pinpoint the exact location of the bug in the code.

c) Bug Resolution

Developers resolve the issue once the bug is isolated and its root cause is identified. Bug resolution involves making the necessary code changes to fix the bug and ensure it no longer

affects the software. Developers modify the source code to correct the identified issue. After making the changes, the code is retested to ensure the bug has been successfully fixed. Additional testing, such as regression testing, may be performed to verify that the fix has yet to introduce new bugs.

2.1.3 Stages of Software Bug

The life cycle of a software bug typically involves several states or statuses, as shown in Figure 2.2, through which the bug progresses as it is identified, addressed, and resolved. Each state represents a specific phase in the bug's life cycle, and developers, testers, and quality assurance teams use these states to track and manage the bug. Here's a description of each of these bug states:

a) New: The bug is newly identified and reported in this initial state. It means that someone, often a tester or user, has noticed a problem or anomaly in the software's behavior and has reported it as a potential bug. The bug still needs to be reviewed or assigned to anyone.

b) Open: After a bug is reported and reviewed, it is typically marked as "open." This status indicates that the reported issue has been acknowledged and is being investigated or worked on by the development or quality assurance team. The responsible team member may analyze the bug to determine its root cause.

c) Assign: Once the bug has been investigated and its root cause is identified, it is assigned to a specific developer or team member responsible for fixing it. The "assign" status signifies that the bug has been allocated to someone for resolution.

d) Deferred: Sometimes, a bug may be deemed less critical or lower in priority, and the decision is made to postpone its resolution to a future release or iteration. In such situations, the bug may be marked as "deferred." Deferred bugs are not addressed immediately but are tracked for future attention.

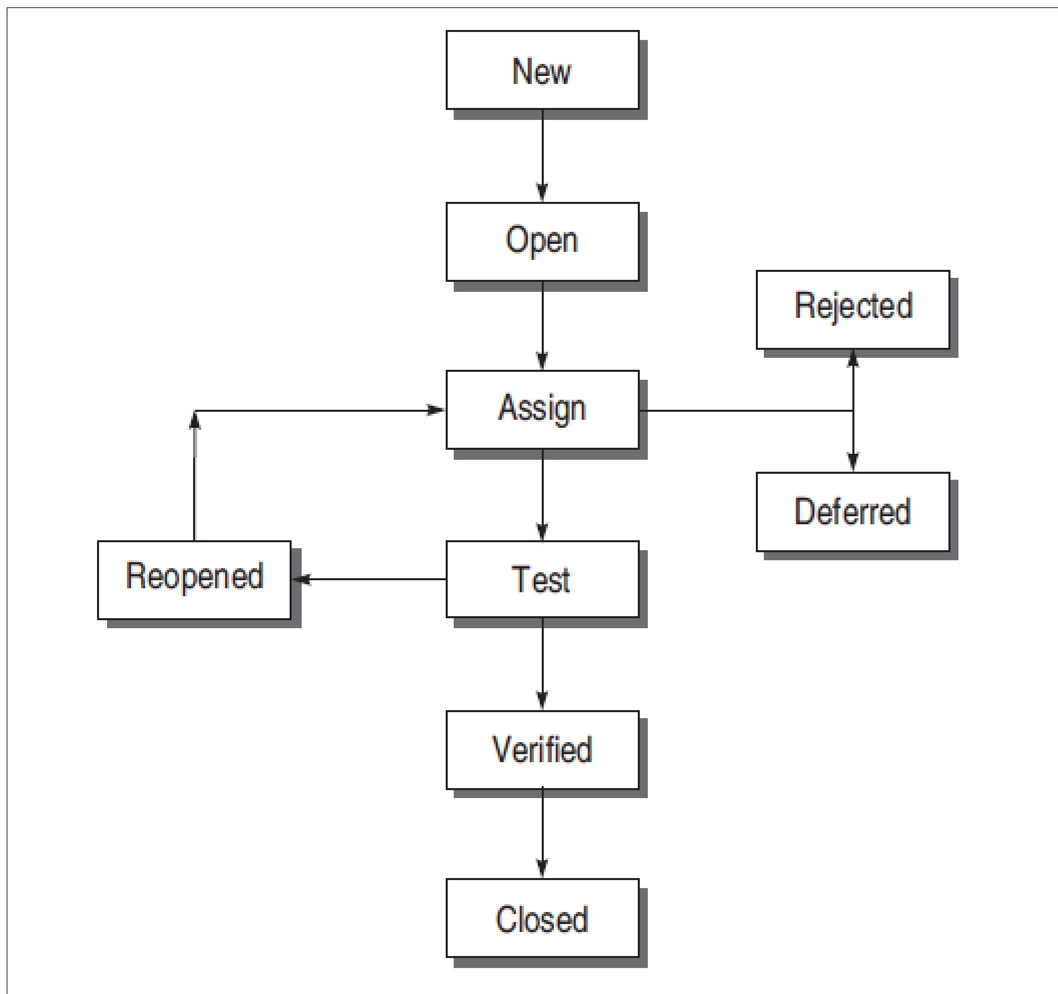


Figure 2.2: Stages of bug [24]

e) Rejected: Occasionally, a reported issue is found not to be a bug, or it may be a duplicate of an existing bug report. In such cases, the bug report may be "rejected." It means that the issue does not require any action, and it is closed without any changes made to the software.

f) Test: After a developer has made code changes to address the bug, the software enters the "test" phase. Testers or quality assurance professionals verify that the bug fix is effective and does not introduce new issues. This phase includes testing the affected functionality to ensure the issue no longer exists.

g) Verified/Fixed: Once the testing phase is complete and the bug fix is confirmed to be effective, the bug is marked as "verified" or "fixed." This status indicates that the reported issue has been successfully resolved, and the code changes are ready for deployment.

h) Reopened: Sometimes, after a bug is marked as "verified" or "closed," it may resurface or reoccur. If the issue reappears, it is reopened, and the development team reevaluates the problem to determine its root cause and resolve it again.

i) Closed: The final stage in the bug life cycle is "closed." A bug is marked as "closed" when it has been successfully fixed, verified, and confirmed as resolved. This status signifies that the bug is considered closed, and no further action is required.

These bug states provide a structured way to manage and track the progress of bug reports throughout the software development life cycle. Effective bug tracking helps ensure that software releases are high quality and free from known defects.

2.1.4 Classification of Software Bug

Software bugs can manifest in various forms, each with unique characteristics and impact on software functionality. Understanding the different types of software bugs is crucial for developers, testers, and quality assurance teams to identify, address, and prevent issues effectively. Bugs are categorized based on various criteria, including:

- Severity
- Nature of the bug
- Platform
- Version
- Component
- Metadata Tagging

Figure 2.3 represents the overall bug classification based on these criteria.

2.1.4.1 Severity of Bugs

The severity of bugs in software development is a critical aspect that categorizes and prioritizes issues based on their potential impact and urgency for resolution. The severity level helps software development teams, testers, and stakeholders assess the importance of addressing a particular bug. The severity of a bug is typically categorized into several groups, each representing a different degree of impact. Table 2.2 represents an overview of expected bug severity levels.

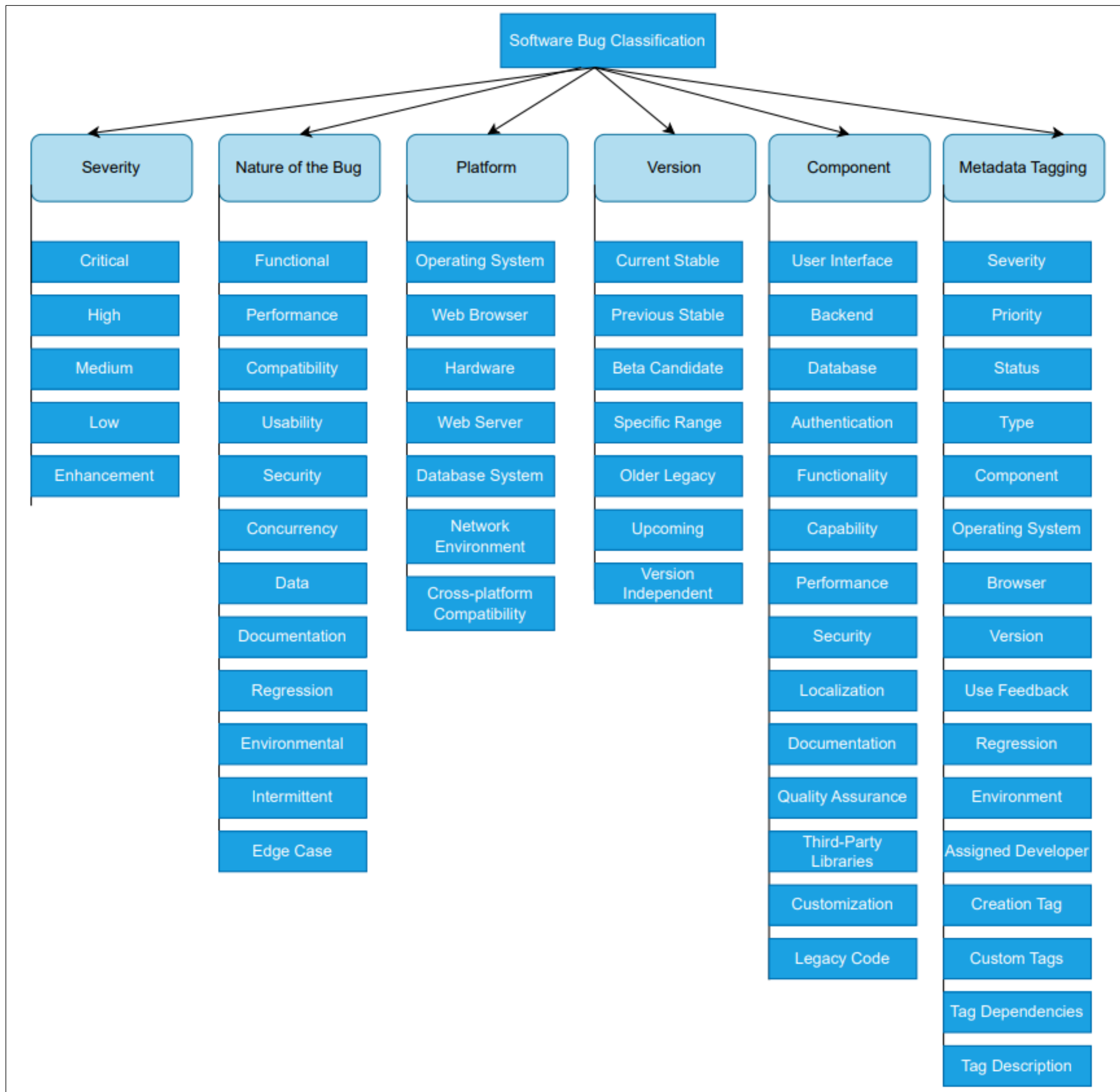


Figure 2.3: Overview of software bug classification

Table 2.2: Classification of bugs based on severity

Bug Severity	Description
Critical	Critical bugs are the most severe and significantly impact the software's functionality. They often result in complete system failure, data loss, security breaches, or other catastrophic consequences. Critical bugs require immediate attention and are typically considered showstoppers that prevent the software from being released or used.
	Continued from next page

Table 2.2– continued from previous page

Bug Severity	Description
High	High-severity bugs substantially impact the software's functionality, but they do not cause system crashes or data loss. They may lead to serious issues, such as incorrect calculations, missing features, or usability problems. High-severity bugs are prioritized for quick resolution but may not require immediate action.
Medium	Bugs categorized as medium severity have a noticeable but moderate impact on the software. They typically involve minor functionality issues or inconveniences that do not critically affect the core operation of the application. These bugs are addressed in the normal development cycle, usually in the next planned release.
Low	Low-severity bugs have minimal impact on the software's functionality and often represent cosmetic or minor issues. They may include typos, minor layout problems, or faulty non-essential features. Low-severity bugs are usually addressed during routine maintenance or included in future releases based on lower priority.
Enhancement	This category does not represent a bug but rather a request for new features or enhancements to existing functionality. These requests are typically not considered defects but are valuable input for future software development efforts. They are prioritized based on their potential benefits and alignment with the product roadmap.

2.1.4.2 Nature of the Bug

Bug classification is an essential aspect of software development and quality assurance, helping teams understand, prioritize, and address issues systematically. Bugs can be classified based on various criteria, including their nature. Table 2.3 illustrates the breakdown of bug classification based on the nature of bugs. Bug classification based on the nature of bugs helps development and QA teams prioritize their efforts, allocate resources effectively, and improve the overall quality of the software. Teams can streamline their bug tracking and resolution processes by categorizing bugs, leading to more reliable and user-friendly software products.

Table 2.3: Classification of bugs based on the nature of the bug

Nature of the Bug	Description
Functional Bugs	These are among the most common types of bugs. Functional bugs occur when the software does not perform as intended according to its functional requirements. This can include incorrect calculations, malfunctioning features, or components that do not work as expected.
Performance Bugs	Performance issues impact the program's speed, effectiveness, and use of resources. These flaws may result in memory leaks, sluggish response times, high CPU consumption, or other problems with performance. Performance issues frequently impact user experience, particularly in resource-intensive apps.
Compatibility Bugs	Compatibility bugs arise when the software does not work correctly with specific hardware, operating systems, browsers, or third-party software. These bugs are crucial to address as they can limit the software's usability across different environments.
Usability Bugs (UI/UX Bugs)	Usability bugs pertain to issues related to the software's user interface (UI) and user experience (UX). These bugs include layout problems, confusing navigation, unclear error messages, and other issues that affect how users interact with the software.
Security Bugs	Security bugs are the most critical type of bug. These vulnerabilities can lead to data breaches, unauthorized access, or other security threats. Common security bugs include code injection vulnerabilities, authentication issues, and data leakage.
Concurrency Bugs	Concurrency bugs occur in multi-threaded or multi-process applications. These bugs result from improper synchronization or race conditions, leading to unpredictable and hard-to-reproduce issues.
Data Bugs	Data bugs involve problems related to data handling and management. These bugs can lead to data corruption, loss, or inaccurate data processing. Data bugs are especially critical in applications that deal with sensitive or mission-critical data.
	Continued from next page

Table 2.3– continued from previous page

Documentation Bugs	Documentation bugs are issues related to the software's documentation, such as user manuals, help files, or API documentation. These bugs can lead to misunderstandings, confusion, and difficulties for users and developers trying to understand and use the software.
Regression Bugs	Regression bugs occur when a previously working feature or functionality stops working as expected after code changes are introduced. These bugs often result from code modifications that unintentionally break existing functionality.
Environmental Bugs	Environmental bugs are specific to the conditions in which the software operates. These conditions can include variations in network connectivity, system configurations, or external dependencies. Environmental bugs may not always be reproducible in all environments.
Intermittent Bugs	Intermittent bugs are particularly challenging to diagnose and reproduce because they occur sporadically and inconsistently. These bugs can be elusive and require extensive testing and debugging efforts to pinpoint and resolve.
Edge Case Bugs	Edge case bugs surface under unusual or uncommon scenarios that may not be encountered during typical usage. Identifying and addressing edge case bugs is essential to ensure the software's robustness.

2.1.4.3 Platform

Bug categorization based on platforms involves classifying reported software bugs into different groups or categories depending on the specific platforms or environments where they occur. This process helps software development teams prioritize and address platform-specific issues effectively. Bugs can be categorized as Table 2.4 based on the platform. Categorizing bugs based on platforms allows development teams to allocate resources more efficiently, as experts with knowledge of the specific platform can focus on resolving the issues. It also helps track the prevalence of bugs on different platforms and make informed decisions about which platforms to prioritize in testing and support. Additionally, platform-

based categorization aids in improving user experience by ensuring that software functions smoothly across various environments, which is particularly important in today's diverse computing landscape.

Table 2.4: Bug classification based on platform

Platform	Description
Operation System	Bugs that manifest only on specific operating systems, such as Windows, macOS, Linux, or mobile OS like Android or iOS. This categorization ensures that the right experts focus on OS-specific problems.
Web Browsers	Bugs specific to particular web browsers like Chrome, Firefox, Safari, or Edge. Given the variations in browser behavior, categorizing bugs by browsers helps in targeted testing and debugging.
Hardware	Bugs related to specific hardware configurations, like graphics cards, processors, or peripherals. Hardware-related categorization is crucial for addressing compatibility issues.
Web Servers	Bugs occur due to differences in web server configurations or software, such as Apache, Nginx, or IIS.
Database Systems	Issues related to database platforms like MySQL, PostgreSQL, or Oracle. Categorizing bugs by database systems is vital for database-driven applications.
Network Environment	Bugs that can occur in specific network environments, such as LAN/WAN, VPN, or proxy configurations.
Cross-platform Compatibility	Bugs that affect multiple platforms and require special attention to ensure cross-platform compatibility.

2.1.4.4 Version

Bug categorization based on software version involves classifying reported software bugs into different groups or categories depending on the specific versions of the software where they are identified. This process aids in managing and prioritizing bug fixes, particularly in software that undergoes frequent updates and releases. Table 2.5 depicts the breakdown of bug categorization by software version. Categorizing bugs by software version helps development teams prioritize their efforts, allocate resources effectively, and communicate with users about the status of reported issues. It ensures that critical problems in the current

stable version receive immediate attention while allowing for the orderly resolution of bugs in older versions. Additionally, it aids in tracking the history of bugs across different software releases, which can provide insights into the software's overall quality and stability over time.

Table 2.5: Bug Classification based on version

Version	Description
Current Stable Version	Bugs are specific to the current stable release of the software. These are typically considered high-priority issues as they affect the most users.
Previous Stable Versions	Bugs that are found in the previous stable versions of the software. These may still require attention, mainly if many users use the previous version.
Beta Candidate Versions	Bugs that occur in pre-release versions of the software, such as beta or release candidate builds. These issues are critical for developers to resolve before the final release.
Specific Version Ranges	Categorizing bugs based on specific version ranges, such as from version 2.0 to 2.5. This allows developers to address issues affecting a particular set of releases.
Older Legacy Versions	Bugs that affect older legacy versions of the software. While these versions may have fewer users, critical bugs in legacy software can still impact a dedicated user base.
Upcoming Versions	Issues discovered in the software's development versions are being prepared for future releases. These bugs are essential to resolve before the next release cycle.
Version-Independent Bugs	Bugs that are not tied to a specific version but are inherent in the software's architecture or design. These are often long-term issues that require extensive redevelopment.

2.1.4.5 Components

Bug categorization based on specific modules or components of software is a crucial aspect of bug management and triaging. It involves classifying reported bugs into categories that correspond to the particular parts or functionalities of the software where they occur. This categorization helps development teams efficiently allocate resources, prioritize bug fixes, and maintain software quality. The detailed breakdown of bug categorization by software

module or component is presented in Table 2.6. Categorizing bugs by a software module or component streamlines bug triaging, allowing development teams to assign bugs to the most relevant experts or teams. It also facilitates tracking and reporting on the status of bugs within different parts of the software, helping ensure that critical issues are addressed promptly while maintaining overall software quality and stability.

Table 2.6: Bug classification based on components

Components	Description
User Interface (UI)	Bugs related to the graphical user interface, including layout, design, responsiveness, and user interaction issues. These can affect the overall usability and user experience of the software.
Backend	Bugs that occur in the backend or server-side components of the software. These may involve data processing, server communication, database interactions, and performance.
Database	Bugs specific to the database layer of the software, such as data corruption, schema issues, query errors, or data retrieval problems.
Authentication	Issues related to user authentication, authorization, access control, and security permissions. These bugs can have significant security implications.
Networking	Bugs affecting network connectivity, data transmission, API integration, or communication protocols. These issues can impact data exchange between software components.
Functionality	Bugs tied to specific software features or functionalities, such as calculations, algorithms, data processing, or specific operations. These can affect core software capabilities.
Compatibility	Bugs related to software compatibility with different platforms, devices, operating systems, browsers, or third-party software. Ensuring compatibility is crucial for a broad user base.

Continued on next page

Table 2.6 – continued from previous page

Performance	Issues that degrade software performance, including slow execution, resource-intensive operations, memory leaks, or bottlenecks. Performance bugs can affect user satisfaction and efficiency.
Security	Bugs with security implications include as vulnerabilities, exploits, code injection, or data breaches. Security-related bugs require immediate attention to protect user data and system integrity.
Localization	Bugs tied to software localization (adapting the software for different languages) and internationalization (making the software globally accessible). These issues involve translations, date formats, and cultural adaptations.
Documentation	Bugs related to user documentation, help files, tooltips, or in-app guidance. Ensuring accurate and helpful documentation is essential for user support.
Quality Assurance	Bugs found in the testing and quality assurance processes, including test case failures, test environment issues, or testing tool problems.
Third-Party Libraries	Bugs stemming from the use of third-party libraries, APIs, or dependencies. Keeping these components up to date and resolving compatibility issues is crucial.
Customization	Issues related to software customization, configuration settings, or user preferences. These can affect how users tailor the software to their needs.
Legacy Code	Bugs associated with older or legacy code sections that may not adhere to current coding standards or practices. Legacy code may require refactoring or updates.

2.1.4.6 Metadata Tagging

Bug categorization based on metadata tagging is a systematic approach to classifying and organizing reported software bugs using metadata attributes. This method enhances bug tracking, management, and prioritization by providing additional context and information about each bug. Table 2.7 depicts an in-depth look at bug categorization through metadata tagging. Software development teams gain a structured and detailed view of reported issues by categorizing bugs with metadata tags. This categorization aids in efficient bug triaging,

assignment, and resolution. It also enables teams to generate reports and dashboards to track bug-related metrics, monitor progress, and make informed decisions about bug fixes and software improvements. Metadata tagging enhances bug management processes and contributes to software quality assurance.

Table 2.7: Bug categorization based on metadata tagging

Metadata Tagging	Description
Bug Severity	Metadata tags can indicate the severity or impact of a bug, such as "Critical," "Major," "Minor," or "Trivial." This helps prioritize bug fixes based on their potential impact on the software and users.
Bug Priority	Tags like "High Priority," "Medium Priority," or "Low Priority" are assigned to indicate the urgency of fixing a bug, considering factors beyond just severity, such as user impact or project deadlines.
Bug Status	Metadata tags denote the current state of a bug in the bug-tracking system. Common statuses include "Open," "Assigned," "In Progress," "Resolved," "Reopened," and "Closed." These tags help track the bug's lifecycle.
Bug Type	Tags specify the type of bug, such as "Functional," "Performance," "Security," "UI/UX," "Compatibility," or "Documentation." Understanding the bug type aids in allocating the right expertise for resolution.
Bug Component	Metadata tags identify the specific software component or module where the bug resides. For instance, "Database," "UI," "Networking," "Authentication," or other relevant component names.
Operating System	Tags indicate the operating system(s) affected by the bug, such as "Windows," "Linux," "macOS," "iOS," or "Android." This information is crucial for understanding platform-specific issues.
Browser	If the bug is related to web applications, tags can specify the affected browsers, such as "Chrome," "Firefox," "Safari," or "Edge." It helps address cross-browser compatibility problems.

Continued on next page

Table 2.7 – continued from previous page

Version	Metadata tags include the software version(s) in which the bug is observed. This assists in version-specific bug tracking and ensures fixes are applied to the appropriate releases.
User Feedback	Tags like "User Reported" or "User Feedback" highlight bugs identified through user reports or feedback channels. These bugs may require additional user testing or verification.
Regression	A "Regression" tag indicates a previously resolved bug reappeared in a newer software version. Recognizing regressions is vital to maintaining software quality.
Environment	Tags describe the specific environmental conditions under which the bug occurs, such as "Production," "Staging," "Development," or "Test." This information aids in reproducing and diagnosing the bug.
Assigned Developer	Tags indicate the developer or team responsible for addressing the bug. This helps distribute bug-fixing tasks among team members.
Creation Date	Tags record the date when the bug report was created. It assists in identifying older, unresolved bugs that may require attention.
Custom Tags	Custom metadata tags can be created to address project-specific attributes or criteria for bug categorization. These tags offer flexibility in organizing bugs based on unique project needs.
Tag Dependencies	Some metadata tags may depend on others. For example, if a bug is tagged as "Security," it may automatically receive a "High Priority" tag.
Tag Descriptions	Tags may include descriptions or guidelines to ensure consistent usage and understanding among team members.

2.1.5 Software Bugs Handling Process

The bug-handling process is crucial to software development and quality assurance, ensuring that identified issues are effectively managed, tracked, and resolved. Figure 2.4 represents an overview of the typical bug-handling process.

i) Bug Identification

The bug-handling process begins with the identification of a potential issue. Bugs can be discovered through various means, including user reports, automated testing, manual testing,

code reviews, or monitoring tools [25].

ii) Bug Reporting

Once a bug is identified, it needs to be reported. A bug report is created, providing essential details about the issue [26]. This report typically includes information such as:

- **Bug description:** A clear and concise description of the problem.
- **Steps to reproduce:** Detailed instructions on how to recreate the bug.
- **Expected behavior:** What should happen when the software is functioning correctly?
- **Actual behavior:** What is observed when the bug occurs.
- **Environment details:** Information about the software environment where the bug was encountered (e.g., operating system, browser, hardware).
- **Severity and priority:** An assessment of how critical the bug is and how urgently it needs to be addressed.

iii) Bug Triaging

Bug triaging is the process of evaluating and prioritizing reported bugs. Bugs are categorized based on severity, impact, and urgency during triage. Each bug is also assigned a priority, indicating how soon it should be addressed. High-priority bugs require immediate attention, while lower-priority bugs may be scheduled for future releases.

iv) Bug Assignment

Once a bug is triaged, it is assigned to a developer or a development team responsible for resolving it. Assigning the bug ensures clear ownership and accountability for its resolution. Figure 2.5 depicts the automated bug assignment process using ensemble-based machine learning models.

v) Bug Tracking

Bug tracking involves using a dedicated bug tracking system or software (e.g., Bugzilla, Jira, or GitHub Issues) to manage and monitor the progress of each bug. Figure 2.6 depicts the bug tracking system of the Team Foundation Server (TFS) on Microsoft Visual Studio. Bug tracking systems allow stakeholders to:

- Track the bug's status (e.g., open, in progress, resolved).
- Record comments, discussions, and updates related to the bug.
- Attach relevant files or screenshots.
- Link the bug to related code changes and development tasks.
- Generate reports and metrics on bug resolution and software quality.

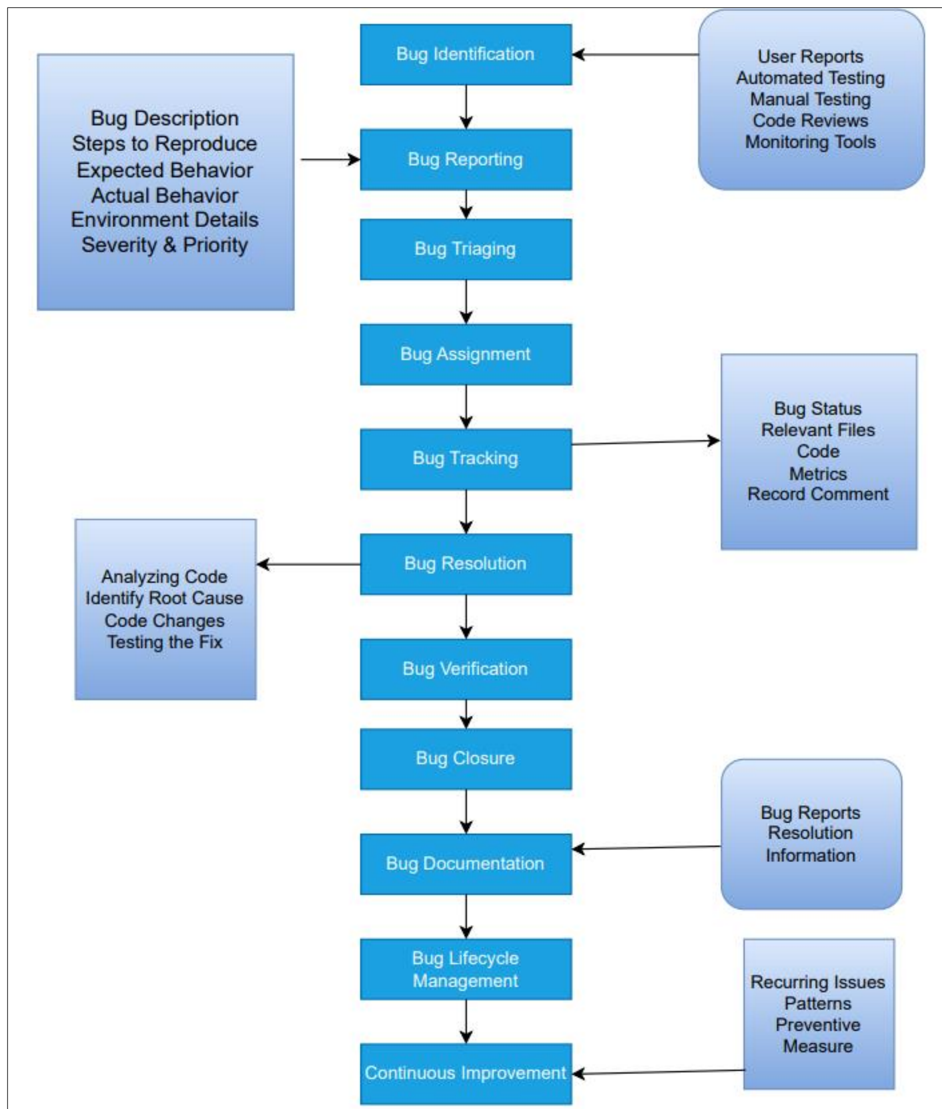


Figure 2.4: Software bug handling process

vi) Bug Resolution

Developers work on resolving the bug based on the information provided in the bug report. This involves analyzing the code, identifying the root cause, making necessary code changes, and thoroughly testing the fix. Once a resolution is implemented, it undergoes a review process to ensure its correctness and effectiveness.

vii) Bug Verification

After a bug is marked as resolved, a verification process is conducted to confirm the issue has been successfully addressed. Testers or QA engineers follow the steps in the bug report to verify that the bug no longer exists in the software.

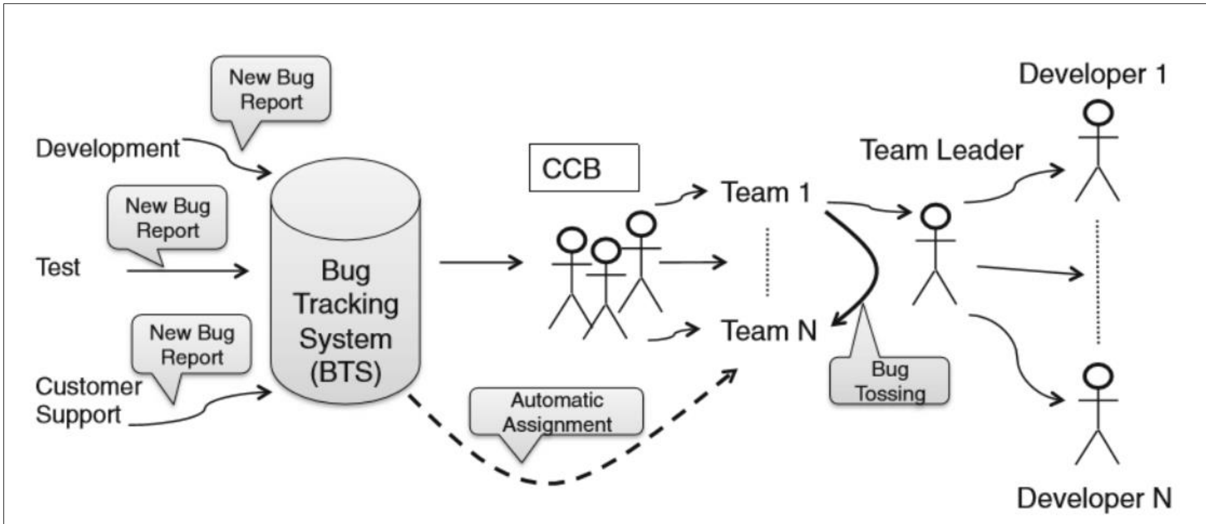


Figure 2.5: Automated bug assignment process [27]

viii) Bug Closure

Once a bug has been verified and confirmed as resolved, it can be marked as closed. Closed bugs are considered officially resolved, and no further action is required unless the issue reoccurs.

ix) Bug Documentation

Proper documentation of bug reports, resolutions, and additional information is essential. Documentation helps in knowledge sharing, future reference, and ensuring transparency in the development process.

Bug 26000: DM Admin cannot see details of "Model" and "Manufacturer" in view pane

Iteration: Deployment Manager\Release\Sprint 19

STATUS

Assigned To:	Developer A
State:	Done
Issue:	Code Defect
Category:	Security
Severity:	3-Normal Bug

DETAILS

Backlog Priority:	1056
Effort:	
Area:	Security
Resolution:	Fixed
Reason:	Defect fixed

Steps to reproduce:

This point cannot be verified with DM Admin Role because user in this role even cannot see this view.

Steps:
1. SM Admin fill all Fields in windows computer form, under the "All Items"
2. Login SM as DM Admin, check all windows computers view.

Expected result:
Project admin can see "Model" and "Manufacturer".
Actual result:
Project admin cannot see "Model" and "Manufacturer".

CHANGE HISTORY:

Developer A deleted a link to Work Item 26868.
Wed, 10/5/2011, 2:35 AM

Developer A changed **Iteration Path** from '**Deployment Manager\Release\Sprint 20**' to '**Deployment Manager\Release\Sprint 19**' and made one other change.
Wed, 10/5/2011, 2:31 AM

Developer B changed **State** from '**Committed**' to '**Done**' and made 3 other changes.
Tue, 9/13/2011, 10:47 AM

Figure 2.6: Bug tracking system of TFS [27]

x) Bug Lifecycle Management

The bug-handling process continues throughout the software development lifecycle, ensuring that newly identified bugs are addressed and resolved bugs do not reappear in subsequent releases.

xi) Continuous Improvement

The bug-handling process should be subject to continuous improvement. Teams should analyze the root causes of recurring issues, identify patterns, and implement preventive measures to reduce the likelihood of similar bugs in the future.

Effective bug handling is critical for maintaining software quality, meeting user expectations, and delivering reliable software products. Following a structured bug-handling process, development teams can efficiently manage and resolve issues, improving software reliability and user satisfaction.

2.2 Bug Reports

A software bug report is a concise and structured document that formally records a detected issue or problem within a software application. It typically includes essential details such as a clear description of the bug, steps to reproduce the issue, the expected and actual behavior, information about the software environment in which the bug was encountered (e.g., operating system, browser), and any relevant attachments, such as screenshots or log files. Bug reports also often assign a severity level and priority to the issue, indicating its impact and urgency. These reports are crucial for developers and quality assurance teams as they provide a foundation for identifying, tracking, prioritizing, and ultimately resolving software defects, contributing to improving software quality and user experience.

2.2.1 Bug Features Description

A bug report is an official document that contains all the essential information about a bug. To report a bug, the tester/QA must complete several features. The team members get a guideline from the bug report to solve the bug. A sample bug report is shown in Figure 2.7. Here are the key features of a bug report in detail:

- **Title/Summary:** The bug report should have a concise and descriptive title or summary that provides a quick overview of the issue. It should be clear and specific, allowing anyone to understand the problem at a glance.
- **Description:** This section provides a detailed account of the bug. It includes information on what the user was doing when the issue occurred, the steps to reproduce the problem, and any error messages or unexpected behaviors observed. The description should be comprehensive and precise.

Figure 2.7: Sample bug reporting form of Bugzilla [28]

- **Environment Details:** Bug reports often include information about the environment in which the bug was encountered. This can encompass details such as the operating system, hardware specifications, software version or build number, web browser (if applicable), and other relevant configurations.
- **Attachments:** Bug reports may include attachments like screenshots, videos, or log files that visually or contextually illustrate the issue. These files help developers better understand the problem and can be crucial for debugging.
- **Severity Level:** Bugs are typically categorized by severity, ranging from critical to minor. This classification indicates the issue's impact on the software's functionality and users. Common severity levels include "critical," "major," "minor," and "cosmetic."

- **Priority:** Priority indicates the bug's urgency in terms of fixing. It helps project managers and developers prioritize which issues should be addressed first. Priorities often include "high," "medium," "low," and "deferred."
- **Assigned To:** In a team or organization, bug reports include a field that designates the developer or team responsible for fixing the issue. This ensures clear accountability.
- **Status:** The status of a bug report tracks its progress through the resolution process. Common statuses include "new," "open," "in progress," "resolved," "verified," "closed," and more. Developers update the status as they work on and complete the bug fixes.
- **Comments/History:** Bug reports maintain a history of comments and changes, documenting all interactions related to the issue. This can include conversations between the reporter, developer, and testers, as well as updates on the bug's status.
- **Additional Information:** Depending on the specific project or organization, bug reports may include additional fields or custom attributes tailored to their needs. These could cover aspects like the component affected, the date reported, or any related issues or dependencies.
- **User Information:** If a user or customer submits the bug report, it may include contact information or details about the user's account, which can be helpful for follow-up or clarification.

A well-structured bug report is a comprehensive document providing all the necessary information for efficiently identifying, reproducing, tracking, and resolving software issues. It is critical in maintaining software quality and ensuring a smooth development process.

2.2.2 Bug Report History

Bug report history is a critical component of software development's bug tracking and management process. It represents a chronological record of all activities, changes, and discussions related to a specific bug or issue reported within a software project. Figure 2.8 illustrates the history information of Bug #456798. The detailed explanation of bug report history is illustrated in Figure 2.9:

Back to ~~bug 456798~~

Who	When	What	Removed	Added
nasz013	2015-03-02 09:27:43 EST	Status	NEW	ASSIGNED
		CC		nasz013
		Assignee	viatra-inbox	nasz013
		Target Milestone	---	0.7.0 M1
nasz013	2015-03-09 05:29:52 EDT	Status	ASSIGNED	RESOLVED
		Resolution	---	FIXED

Back to ~~bug 456798~~

Figure 2.8: Bug report history [29]

- **Creation:** The bug report history typically begins with the creation of the report. This marks when a user or tester identifies a problem within the software and submits a formal bug report. The initial details provided, including the bug's title, description, and any attached files, are captured as part of the report's history.
- **Assignment:** Once the bug report is created, it is usually assigned to a developer or a development team responsible for addressing the issue. This assignment is recorded in the bug report history and indicates who is accountable for resolving the bug.
- **Status Changes:** As the developer starts working on the bug, the bug report's status changes. Common status transitions include "new," "open," "in progress," "resolved," "verified," and "closed." Each status change is recorded in the bug report's history, providing a clear timeline of the bug's progress.
- **Comments and Discussions:** Bug reports often facilitate communication between team members and stakeholders. Comments and discussions related to the bug are recorded in the history, allowing team members to collaborate, share insights, and exchange information about the issue. These comments may contain technical details, proposed solutions, or additional context.
- **Attachments:** If additional files, such as screenshots, log files, or test cases, are attached during the discussion or investigation of the bug, these attachments become part of the bug report's history. They serve as visual or contextual aids for understanding and resolving the issue.
- **Changes in Severity or Priority:** In some cases, the severity or priority of a bug may change as more information becomes available or the team gains a deeper

understanding of its impact. Such changes are documented in the bug report's history, and reasons for these alterations may be provided.

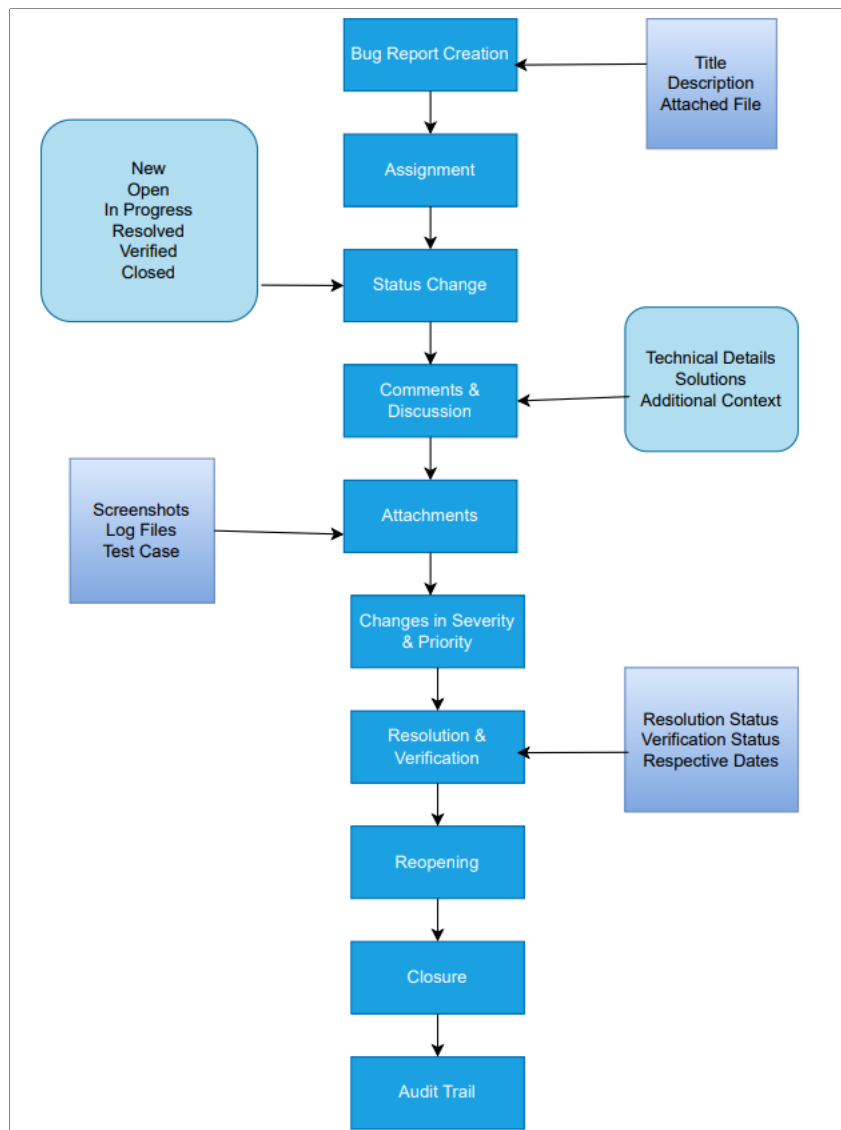


Figure 2.9: Bug history creation process

- **Resolution and Verification:** When developers believe the bug is fixed, they mark it as "resolved." The bug is then verified by testers or quality assurance personnel to confirm that the issue is indeed resolved. Both the resolution and verification statuses, along with their respective dates, are recorded in the history.
- **Reopening:** If the bug persists or reoccurs after being marked as "resolved" and subsequently "verified," it may be reopened. This action is documented in the bug report's history and the process of addressing the bug restarts.
- **Closure:** Finally, when the bug is successfully resolved and verified, it is marked as "closed." The closure date and any additional notes regarding the resolution are

documented in the history.

- **Audit Trail:** Bug report histories serve as audit trails, providing a transparent and comprehensive account of all activities related to a particular bug. This audit trail is invaluable for tracking changes, identifying bottlenecks, and ensuring accountability throughout the bug resolution process.

2.3 Bug Triaging for Developer Recommendation

Bug triaging is a critical process in software development and bug management that involves the assessment, classification, and prioritization of bug reports or issues submitted by users, testers, or automated testing tools. This process is essential for effectively managing the influx of bug reports and ensuring that development teams can efficiently allocate resources to address the most critical issues. The detail of the bug triaging process is described in Figure 2.10.

2.3.1 Bug Submission

Bug submission, or bug reporting or issue reporting, is the first and crucial step in the bug triaging process. It involves users, testers, or stakeholders reporting identified software defects or issues to the development team for resolution. The bug submission process commences when a user, tester, or any individual engaged with the software encounters an issue or abnormal behavior while using the application. This issue could range from functional errors, crashes, performance issues, usability concerns, or security vulnerabilities. The person encountering the problem identifies and acknowledges the issue. They recognize that the observed behavior is unintended, undesirable, or contrary to the expected functionality of the software. The person documenting the bug compiles detailed information about the issue they have encountered. This documentation aims to provide developers with a comprehensive understanding of the problem, ensuring efficient resolution. The information typically includes:

- **Bug Description:** A clear and concise description of the issue, highlighting the problem's nature and impact on the software's functionality or user experience.
- **Steps to Reproduce:** A step-by-step account of the actions taken before and during the bug occurrence. This is crucial for developers to recreate the issue in their testing environment.
- **Environment Details:** Information about the user's configuration, including the operating

system, hardware specifications, software versions, and any relevant settings or configurations.

- **Attachments:** Visual aids, if applicable, such as screenshots, videos, or log files that provide additional context or evidence of the bug.
- **Expected vs. Actual Behavior:** A clear comparison between what the user expected to happen and what occurred. This helps developers understand the deviation from the intended functionality.

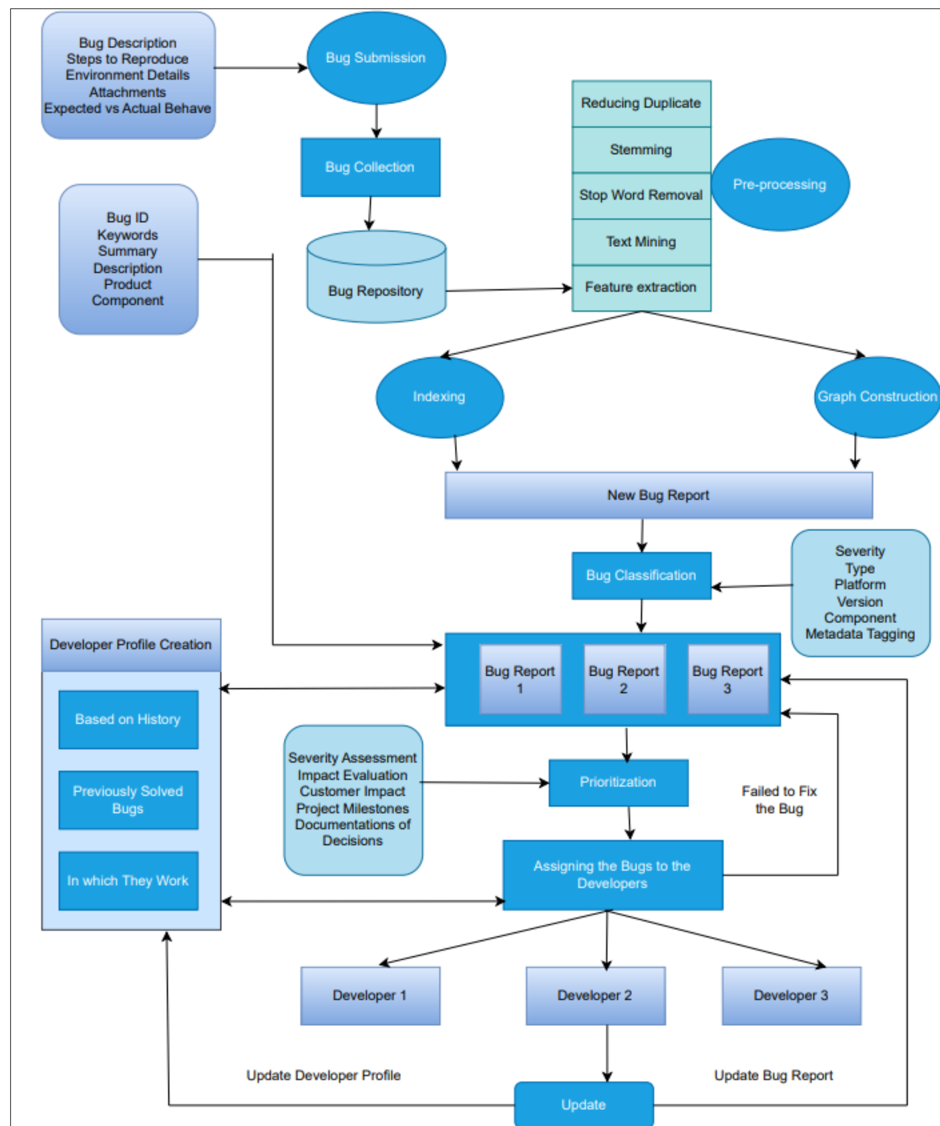


Figure 2.10: Bug triaging process for developer recommendation

The documented bug report is typically submitted through a dedicated bug tracking system or issue management tool, such as Bugzilla, Jira, GitHub Issues, or similar platforms. These systems provide structured forms for bug submission and ensure that relevant information is captured. Users or testers access the bug tracking system and fill out a bug

submission form. This form prompts them to enter the essential details mentioned earlier, ensuring the bug report is well-documented. Once the bug report is submitted, the system generates a unique identifier or bug report number. This identifier is used to track the bug's progress throughout the triaging and resolution process. After submission, the bug report goes through an initial review process. A bug triage team or a designated triager typically conducts this review. This review aims to ensure that the bug report is complete, clear, and contains all necessary information for further assessment.

2.3.2 Data Collection and Pre-processing

The data collection process in bug triaging serves as the cornerstone for the entire bug resolution workflow. It encompasses a series of well-defined steps to assemble a comprehensive and structured dataset of bug reports. Initially, bug report sources are identified, typically including bug tracking systems like Bugzilla, GitHub Issues, or Jira. These sources are repositories of reported software issues and are accessed using various methods such as web scraping, APIs, or data exports. Once the data is retrieved, it undergoes careful filtering to isolate relevant bug reports based on criteria like project, software version, date range, or specific keywords. Subsequently, an essential phase of data preprocessing is executed, which involves cleaning the text to eliminate irrelevant or sensitive information, thus ensuring data integrity. The bug reports are then structured and organized into a database or structured file format, ensuring that each report includes critical details such as a unique identifier, bug description, status, severity, priority, assigned developer, and timestamps. For enhanced context and analysis, bug reports may be enriched with supplementary information, such as developer profiles or commit history. Data indexing is also imperative for swift access to relevant bug reports during the bug-triaging process. Sometimes, a graph structure is used to show how bug reports, developers, and other things are connected. A versioning system monitors bug report changes over time, facilitating historical data referencing and maintaining data accuracy. Finally, regular updates to the dataset are essential to keep it current and reflective of the evolving software environment.

The integrity and comprehensiveness of the bug report dataset are paramount to the success of bug-triaging systems. A meticulously designed and executed data collection pipeline ensures that bug reports are systematically gathered, structured, and maintained. This foundational process empowers bug-triaging systems to function effectively by providing developers with the necessary information to accurately assign and prioritize bug resolution

tasks. Additionally, it supports more advanced bug-triaging tasks like developer recommendation and load balancing by offering valuable insights into bug characteristics, historical trends, and developer expertise. Consequently, a well-documented and robust data collection strategy is instrumental in enhancing the efficiency and accuracy of bug resolution efforts in software development projects.

The commit logs and bug reports consist of narrative text formats. Additionally, the source entities are constructed by merging keywords. Therefore, it becomes necessary to perform preprocessing on all these sources of information to eliminate duplicate and irrelevant terms. Below, we outline some frequently employed preprocessing procedures in the automated bug assignment process.

2.3.2.1 Stemming

Stemming is a linguistic normalization technique used in text processing to reduce words to their root or base form. It involves removing suffixes from words to convert them into their core form [30, 31]. This can be particularly helpful in text analysis when you want to group words that share a common root. For instance, after stemming, words like "jumping," "jumps," and "jumped" would all be reduced to the common root "jump." Table 2.8 represents the example of stemming.

Table 2.8: Example of stemming

Original Words	After Stemming
Jumping	Jump
Jumps	Jump
Jumped	Jump

In this example, the words have stemmed from their base form, "jump," making it easier to analyze and categorize them based on their shared root. Stemming is often used in natural language processing tasks like text classification, information retrieval, and sentiment analysis to reduce text data's dimensionality and improve text-based algorithms' efficiency. Researchers develop various stemming algorithms to convert the inflectional form of keywords towards their root form. A brief description of these algorithms is described in Table 2.9.

Table 2.9: Algorithms of stemming

Algorithms	Description	Example
Porter Stemmer [32]	The stemming method, created by Martin Porter in 1980, is among the most popular and established. Uses a series of rules and transformations to reduce words to their base or root form.	"Jumping," "Jumps," "Jumped" becomes "Jump."
Snowball Stemmer (Porter2) [33]	An improvement over the original Porter Stemmer. Designed for multiple languages and offers more accurate stemming for them.	"Running," "Runs," "Run" becomes "Run."
Lancaster Stemmer [34]	Developed by Chris Paice in 1990. Known for its aggressive stemming, often resulting in very short stems	"Maximum," "Maximize," "Maximization" becomes "Maxim."
Lovins Stemmer [35]	Developed by J. P. Lovins in 1968. Focuses on maintaining word readability while stemming.	"Flights," "Flight's," "Flights'" becomes "Flight."
Paice/Husk Stemmer [36]	Developed by Chris Paice, also known as the Husk stemmer. Balances aggressiveness with maintaining readability.	"Dogs," "Dog's," "Dogs'" becomes "Dog."
Krovetz Stemmer [37]	Developed by Robert Krovetz. Known for its stemming of complex words.	"Easily," "Ease," "Eases" becomes "Easili."

These stemming algorithms are used to reduce words to their root forms, which can help in various natural language processing tasks like text classification, information retrieval, and text analysis. The choice of algorithm depends on your specific language and application requirements.

2.3.2.2 Stop Word Removal

In natural language processing and text analysis, stop-word removal is crucial in refining textual data for subsequent analysis and machine learning tasks. Stop words encompass common, non-content-bearing words such as "the," "and," "in," "of," and many others that appear frequently in natural language text but do not carry substantial semantic meaning. The primary objective of stop word removal is to cleanse text data by eliminating these ubiquitous and often superfluous terms, thus enhancing the quality and relevance of the remaining content.

Eliminating stop words makes the text more concise and emphasizes the importance of keywords and phrases. This streamlining process not only aids in reducing computational complexity but also contributes significantly to the accuracy and efficiency of downstream NLP algorithms. It allows subsequent tasks like text classification, sentiment analysis, information retrieval, and topic modeling to operate on more salient and informative text, enabling them to discern meaningful patterns, relationships, and insights.

Stop word removal finds widespread application across various domains, from information retrieval systems, where it helps optimize search queries and document retrieval, to sentiment analysis, ensuring that sentiment-bearing words are prominent in sentiment scoring. Moreover, in topic modeling, eliminating stop words contributes to identifying coherent themes and topics within textual corpora. Overall, stop word removal is a fundamental pre-processing step in NLP, ultimately improving NLP applications' quality, efficiency, and interpretability.

2.3.3 Indexing

In bug triaging, indexing refers to creating a structured and organized database or index of bug reports and related information. This index is essential for efficient bug management and allocation. During indexing, various attributes and metadata associated with each bug report are extracted, parsed, and stored in a way that allows for quick and effective retrieval and analysis.

Table 2.10: Different indexing process

Indexing	Description	Usage
Keyword-Based Indexing [38]	This approach indexes bug reports based on specific keywords, terms, or phrases within the report's textual content.	It allows users to search for bug reports by entering relevant keywords or terms, making it a simple and effective way to find relevant reports.
Metadata-Based Indexing [39]	Bug reports often contain metadata such as bug IDs, timestamps, severity levels, and assignees. Metadata-based indexing focuses on indexing bug reports using these metadata attributes.	It allows users to search for bug reports by entering relevant keywords or terms, making it a simple and effective way to find relevant reports.
Full-Text Indexing [40]	Full-text indexing involves creating an index of the entire textual content of bug reports, including descriptions, summaries, and comments.	This approach enables comprehensive searching, allowing users to perform text-based queries across all bug report content.
Concept-Based Indexing [41]	Concept-based indexing relies on natural language processing and semantic analysis to identify and index the concepts or topics discussed within bug reports.	Users can search for bug reports based on the underlying concepts, making it effective for identifying reports related to specific issues or topics.
Vector Space Model (VSM) [42]	VSM indexing represents bug reports and related documents as vectors in a multidimensional space. Terms and keywords are assigned weights, and the similarity between bug reports and queries is measured based on vector angles.	VSM indexing allows for ranked retrieval of bug reports, helping users find the most relevant reports based on query relevance.
Continue on the next page		

Table 2.10– continued from previous page

Term Frequency-Inverse Document Frequency (TF-IDF) [43]	TF-IDF indexing calculates the importance of terms within bug reports relative to their frequency across the entire dataset. Terms with higher TF-IDF scores are considered more relevant.	It is used for ranking bug reports based on the importance of terms, helping users identify reports that contain significant keywords.
Graph-Based Indexing [44]	Graph-based indexing represents bug reports and their relationships as nodes and edges in a graph. Nodes can represent bug reports, while edges denote relationships, such as duplicates or dependencies.	This approach is valuable for visualizing and exploring relationships between bug reports, aiding in identifying dependencies or duplicate issues.
Time-Based Indexing [45]	Time-based indexing focuses on indexing bug reports based on timestamps or historical data. It allows users to track the evolution of bugs over time.	This approach is useful for analyzing trends, identifying recurring issues, and assessing the progress of bug resolution efforts.

Bug reports contain a wealth of information, including bug descriptions, summaries, comments, severity levels, timestamps, and more. During indexing, these pieces of information are extracted and separated into distinct fields for systematic storage and retrieval. Bug data may come in various formats and styles, so normalizing the data into a consistent structure is essential. This step ensures that data is uniform and can be easily compared and analyzed. The textual content within bug reports is divided into individual tokens or words. Tokenization helps in creating an organized representation of the textual data. Commonly occurring stop words, such as "the," "and," and "is," may be removed during indexing to reduce noise and improve the relevance of the indexed data. Stemming algorithms may be applied to reduce words to their root form. For example, "running," "ran," and "runner" may all be stemmed from "run." This simplifies the indexing process and allows for more flexible searching. An index is created once the bug report data is prepared and structured. This database index maps keywords, terms, or tokens to the bug reports containing them. It enables efficient searching and retrieval of relevant bug reports based on specific

criteria. Necessary metadata such as bug IDs, timestamps, assignees, and bug statuses are also indexed along with textual data. This metadata is crucial for sorting, filtering, and prioritizing bug reports. The indexing system should provide query capabilities, allowing users to search for bug reports based on keywords, attributes, or criteria. This is a fundamental feature of bug-triaging systems. Several indexing techniques and approaches are used in bug-triaging systems. Table 2.10 represents some different indexing processes commonly employed.

2.3.4 Graph Construction

During the process of resolving bug reports, numerous developers often engage in communication and collaboration. Developers possess expertise in addressing various types of bugs, and some software components may contain a larger number of bugs compared to others. To make the most of this information when suggesting ultimate bug fixers, different methodologies are employed to create graphs based on source code and bug history. For instance, Figure 2.11 illustrates a Developer-Component-Bug (DCB) sample, utilized in [46] to extract developer expertise details.

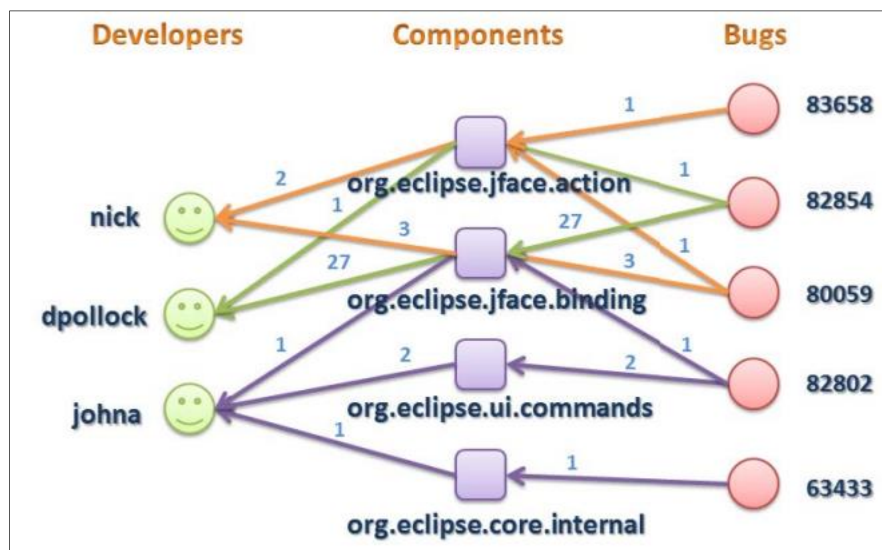


Figure 2.11: Sample Developer-Component-Bug network [46]

The directed DCB network in Figure 2.11 is constructed using three distinct node types: Developer, Component, and Bug. The Developer nodes represent the actual bug fixers involved in resolving the reports. Component nodes signify sets of source files developers modify, while Bug nodes represent fixed bugs. The edges within the graph depict the relationships between developers and the source code components they have collaborated on.

Additionally, the connections between components and their associated bugs indicate which files were modified during bug resolution. These graph structures identify past collaborations among developers, and these relationships can be harnessed in bug assignment processes to identify suitable bug-fixing teams.

2.3.5 Categorization

Bugs are categorized based on various criteria, including:

- **Severity:** The degree of impact of the bug on the software. Common severity levels include critical, major, minor, and trivial.
- **Type:** The category or nature of the bug (e.g., functional, usability, performance, security).
- **Platform:** The specific operating system, device, or environment where the bug occurs.
- **Version:** The software version in which the bug is reported.
- **Component:** The specific module or component of the software affected by the bug.
- **Metadata Tagging:** Metadata can be included in each bug report for future searches and analysis.

A detailed description of bug classification is described in Section 2.1.3.

2.3.6 Prioritization

Bug triagers prioritize bugs to determine the order in which they should be fixed. Priority is based on factors such as:

Severity Assessment: One of the primary factors considered when assigning priority is the bug's severity. Severity indicates the impact of the bug on the software and its users. Typically, bugs are categorized into several severity levels, such as Critical, Major, Minor, and Trivial. Critical issues, which can cause system failures, data loss, or security vulnerabilities, are assigned the highest priority. In contrast, trivial issues, which may result in cosmetic or non-essential problems, are given prioritized less.

Impact Evaluation: Bug triagers assess the overall impact of the bug on the software system. Bugs that affect a significant portion of the user base or impact critical functionalities are prioritized higher. These bugs have the potential to disrupt the user experience or business operations and require immediate attention.

Customer Impact: Bugs reported by key customers, stakeholders, or clients who significantly influence the development process may receive special consideration. Customer-reported issues are often prioritized to ensure customer satisfaction and maintain a positive relationship.

Project Milestones: Bug triagers take into account project milestones and release schedules. Bugs that block upcoming releases or critical project milestones are assigned a higher priority. Addressing these bugs becomes a top priority to ensure the project stays on track and meets its deadlines.

Documentation of Decisions: To maintain transparency and provide a clear rationale for priority assignments, bug triagers document their decisions. They create records that outline the criteria used for prioritization, the specific factors influencing the decision, and any supporting evidence. This documentation helps development teams understand why certain bugs were prioritized and facilitate effective communication among team members.

2.3.7 Assignment and Notification

In some bug-triaging systems, predefined assignment rules or algorithms may be used to automate the assignment process. These rules can be based on bug type, component, developer workload, and historical assignment patterns. For example, a law might specify that all critical security bugs are automatically assigned to a designated security expert. The availability of developers also plays a role in assignment. Triagers consider the workload of potential assignees. Developers who are already overloaded with tasks may not be the best choice for a new assignment. Balancing workload is essential to prevent burnout and ensure timely bug resolution. The bug triaging team may review developers' past performance and track records when assigning bugs. Developers with a history of successfully resolving similar issues or a strong track record of bug fixes may be preferred. Collaboration and communication among developers are vital in resolving complex bugs. Bug triagers may assign bugs to teams or individuals with a history of effective collaboration, ensuring that communication channels are established. Once the bug is assigned to a developer or team, all relevant bug report details are provided to them. This includes the bug description, steps to reproduce, any attached files or screenshots, and any additional context or information gathered during triaging. Comprehensive bug report details are essential to ensure that the developer has a clear understanding of the issue.

Developers are informed about the bug's priority and severity. This information helps them understand the task's urgency and whether it requires immediate attention. Developers receive context about why they were chosen for the assignment. This may include information about their expertise in the relevant area, their past bug-fixing history, or their team's specialization. Assignee notification is often integrated with the bug-tracking system used by the development team. This integration ensures developers can access the bug report directly from the tracking system, update its status, and communicate with other team members. Developers are made aware of their responsibilities regarding the assigned bug. This may include setting a target resolution date, communicating progress, and collaborating with other team members or stakeholders. Assignee notification also establishes a feedback loop. Developers can acknowledge the assignment, seek clarification, and report when the bug is resolved or requires further attention.

2.3.8 Update Developer and Bug Profile

Developer profiles are continuously updated based on the developer's performance in resolving bugs. When a developer successfully fixes bugs, their profile may be updated to reflect their proficiency in handling specific types of bugs. However, weaknesses may be identified in their profile if a developer consistently struggles with certain bug types. These updates can help improve bug allocation and developer recommendations.

Bug profiles are also updated over time. As more information becomes available about a bug, such as its severity, priority, and resolution status, its profile may change. For example, if a bug is initially categorized as low severity but later identified as critical, its profile will be updated to reflect this change. These updates ensure the bug triaging process considers the most current information when making bug assignments.

2.4 Load Balancing in Bug Triggering

Load balancing in the context of bug triggering refers to the distribution of bug-triaging tasks among developers or development teams in a way that optimizes resource utilization, maximizes efficiency, and ensures that bugs are addressed promptly. Effective load balancing can significantly impact the bug-triaging process by preventing bottlenecks, reducing response times, and improving software quality. Figure 2.12 explores the details of load balancing in bug triggering:

i) Task Allocation: Load balancing starts with the allocation of bug triaging tasks to developers or teams. This allocation should consider several factors, including:

- ***Developer Expertise:*** Assign bugs to developers with the relevant expertise and domain knowledge. Experienced developers may handle complex issues, while more straightforward bugs can be allocated to junior developers.
- ***Workload:*** Assess the existing workload of developers. Avoid overloading a single developer with too many tasks, as it can lead to burnout and delays.
- ***Bug Severity:*** High-priority bugs should be assigned promptly, while lower-priority bugs can be scheduled accordingly.
- ***Developer Availability:*** Consider developers' availability and working hours to ensure that assigned tasks can be addressed in a timely manner.

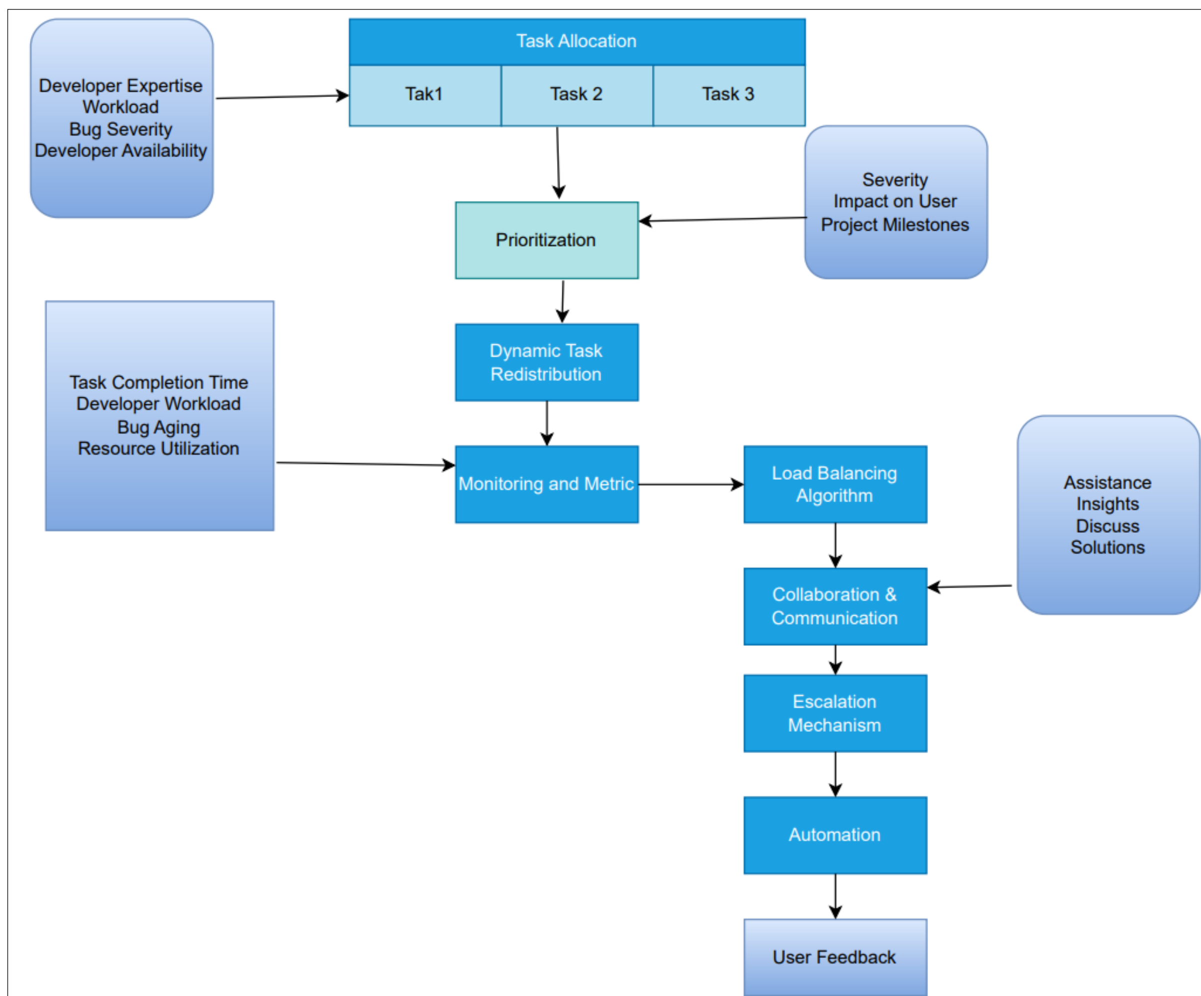


Figure 2.12: Load balancing in bug triaging

ii) Prioritization: Bugs should be prioritized based on their severity, user impact, and project milestones. Load balancing should take into account the priority of bugs when allocating

tasks. Critical bugs may need immediate attention, while less severe issues can be scheduled for later.

iii) Dynamic Task Redistribution: Load balancing is not a one-time activity but an ongoing process. As bug triaging progresses, new bugs arrive, developer's complete tasks, and priorities may change. Load balancing mechanisms should be dynamic and adaptable to redistribute tasks as needed.

iv) Monitoring and Metrics: To achieve effective load balancing, bug-tracking systems should provide monitoring and metrics capabilities. Key metrics to consider include:

- ***Task Completion Time:*** Measure the time developers take to resolve bugs. Identify bottlenecks and delays in the triaging process.
- ***Developer Workload:*** Track the number of tasks assigned to each developer. Ensure a balanced distribution of tasks.
- ***Bug Aging:*** Monitor how long bugs remain open. Older bugs may require reassignment or escalated priority.
- ***Resource Utilization:*** Assess how efficiently resources (developers) are utilized. Avoid situations where some developers are idle while others are overloaded.

v) Load Balancing Algorithms: Various load balancing algorithms can be employed, including round-robin assignment, least busy developer assignment, or algorithms considering both the developer's expertise and workload. The choice of algorithm should align with the bug-tracking system's requirements and goals.

vi) Collaboration and Communication: Load balancing should encourage collaboration among developers. Developers should be able to communicate with each other to seek assistance, share insights, and discuss solutions. Collaboration can lead to more efficient bug resolution.

vii) Escalation Mechanisms: In cases where a bug cannot be resolved within a reasonable timeframe or by the assigned developer, load balancing should include escalation mechanisms. This involves reassigning the bug to a more qualified developer or team.

viii) Automation: Some bug-tracking systems incorporate automation for load balancing. For example, machine learning models can predict task completion times and suggest optimal task assignments.

ix) User Feedback: User feedback and bug reports can play a role in load balancing. If a particular bug affects a large user base, it may receive higher priority and be assigned to a team with the necessary resources to address it promptly.

Effective load balancing in bug triggering ensures that bugs are triaged and resolved efficiently, reducing software downtime, improving user satisfaction, and enhancing overall software quality. It requires continuous monitoring, careful task allocation, and collaboration among developers to achieve optimal results.

2.5 Summary

In today's software development landscape, the effective management of software bugs has emerged as a pivotal endeavor to ensure software systems' sustained quality and reliability. Automatic bug triaging, a vital component of this management process, has assumed a central role in allocating bug-fixing tasks to the right developers. Within this chapter, we embark on a journey to lay the robust conceptual foundation required for a comprehensive understanding of the proposed bug-triaging technique. Our exploration begins with an exhaustive examination of software bugs, unraveling the intricate web of factors contributing to their occurrence and delving into their life cycle. We explore their origins, understanding why they surface and how they evolve through different phases, from their initial detection to final resolution. This comprehension is crucial in navigating the intricate bug-triaging landscape. Additionally, we delve into the multifaceted realm of bug classification, where we decipher the diverse nature of software bugs. Understanding the taxonomy of bugs based on their characteristics and impact is pivotal in ensuring that the right experts are assigned to resolve them. As we traverse further into the bug-handling process, we unveil the intricate steps in identifying, reporting, and managing software bugs. This process, rife with challenges and intricacies, is a linchpin in the bug resolution journey. Furthermore, we provide a comprehensive overview of the bug reporting process, shedding light on bug reports' various features and facets. Bug reports, the vital vessels of bug-related information, are meticulously dissected to reveal their essential components, which include severity assessments, prioritization, and detailed descriptions of the issues encountered. To cap this foundational chapter, we introduce the overarching architecture that underpins automated bug triaging, explicitly focusing on developer recommendation and load balancing. This architectural blueprint sets the stage for the subsequent chapters, where we delve into detailed discussions and present empirical findings illuminating the practical aspects of our bug-triaging approach. By establishing this solid conceptual groundwork, we pave the way for a more comprehensive and insightful exploration of the bug-triaging landscape.

Chapter 3

Literature Review

Automatic bug triaging for developer recommendation and load balancing is crucial in modern software development for several reasons: It significantly improves the efficiency of bug resolution by rapidly and accurately assigning bugs to developers with the right skills and expertise, thereby reducing the time and resources required to maintain software quality. Furthermore, it ensures optimal resource allocation by preventing experienced developers from overloading with excessive bug assignments while providing opportunities for less experienced developers to learn and contribute effectively. Automatic bug triaging also aids in prioritization, identifying critical issues that require immediate attention, and mitigating their impact on software functionality. Additionally, it promotes knowledge sharing among team members, as it encourages less experienced developers to learn from their more experienced peers, ultimately elevating the team's overall skill level. By reducing delays, errors, and bottlenecks in the bug resolution process, automatic bug triaging streamlines the software development cycle, leading to enhanced job satisfaction among developers and a more productive and collaborative work environment.

Having delved into the foundational concepts of software bug management and the framework underpinning automated bug triaging in the preceding chapters, we now focus on the broader landscape of existing research and practices. This Literature Review section explores prior works, methodologies, and techniques that have contributed to the evolution of bug-triaging processes. By scrutinizing the literature, we seek insights into the prevailing challenges, innovative solutions, and emerging trends in bug triaging. Our aim is to leverage the knowledge and experiences gleaned from the past to inform and enhance the bug-triaging approach presented in this study. Through an in-depth analysis of relevant studies and methodologies, we endeavor to build upon the solid foundation established in the introductory chapters and contribute to advancing bug management practices.

3.1 Existing Research Works

Many research endeavors have suggested suitable developers for resolving recently reported bugs in this domain. Most existing techniques rely on the developers' past bug-fixing track record to make these recommendations. In this section, we emphasized the various bug-triaging methods that have significantly impacted shaping our approach. To facilitate understanding, we have classified prior investigations into eleven distinct areas, each aligning with specific aspects of our approach. These categories are denoted as follows:

- Topic model-based approach
- Information retrieval-based approach
- Social network analysis-based approach
- Dependency-based approach
- Machine learning-based approach
- Reassignment-based approach
- Text categorization-based approach
- Data reduction-based approach
- Cost aware-based approach
- Industry oriented-based approach

In the following sections, each of these approaches is described in detail.

3.2 Topic Model-Based Approach

Topic model-based approaches in bug triaging involve using probabilistic models to identify latent topics or themes within a collection of bug reports. These approaches aim to discover the underlying topics that can help understand the content and characteristics of bug reports more effectively.

3.2.1 DRETOM

Xie et al. [47] introduced a topic model-based approach called DRETOM (Developer Recommendation based on Topic Models). DRETOM recommends developers based on their bug-fixing history. The approach comprises multiple steps:

In Step 1, the authors employ the Topic Modeling Toolbox (TMT) with Latent Dirichlet Allocation (LDA) to create topics from existing bug reports. Each bug is associated with the topic with the highest probability in its related bug report. In Step 2, relationships

between developers and bug topics are established using a probabilistic model that calculates a developer's bug-solving probability based on their interest and expertise. Step 3 involves recommending that developers address new bug reports. When a new bug report arrives, it is assigned a topic based on previously built topic models. The probability of each developer being a candidate to fix the bug is expressed as a conditional probability, P(dev | bug). Developers are then ranked based on this probability, and the top K developers are recommended. However, it's important to note that DRETOM has limitations, as it cannot recommend new developers.

3.2.2 BUTTER

Zhang et al. [48] proposed BUTTER, a bug triage approach that utilizes topic modeling and heterogeneous network analysis to automate bug assignments. BUTTER aims to assign bugs to developers capable of solving them. Figure 3.1 shows the overall procedure of BUTTER. The approach leverages both textual and structural information from bug reports. The process begins with applying the LDA algorithm to categorize textual data from previously resolved bugs into various topics. Bug reports contain both textual and structural data, and heterogeneous networks are used to gather structural data about developers. The RankClass model is trained based on the topic model and structural data derived from bug reports. Upon the arrival of a new bug report, it is submitted to the LDA and RankClass models to classify its final topic distribution. The final topic distribution and developer skills in the relevant topic determine the list of developers suitable for addressing the bug. Despite its efficiency gains compared to methods relying solely on textual content, BUTTER shares the limitation of not being able to recommend new developers.

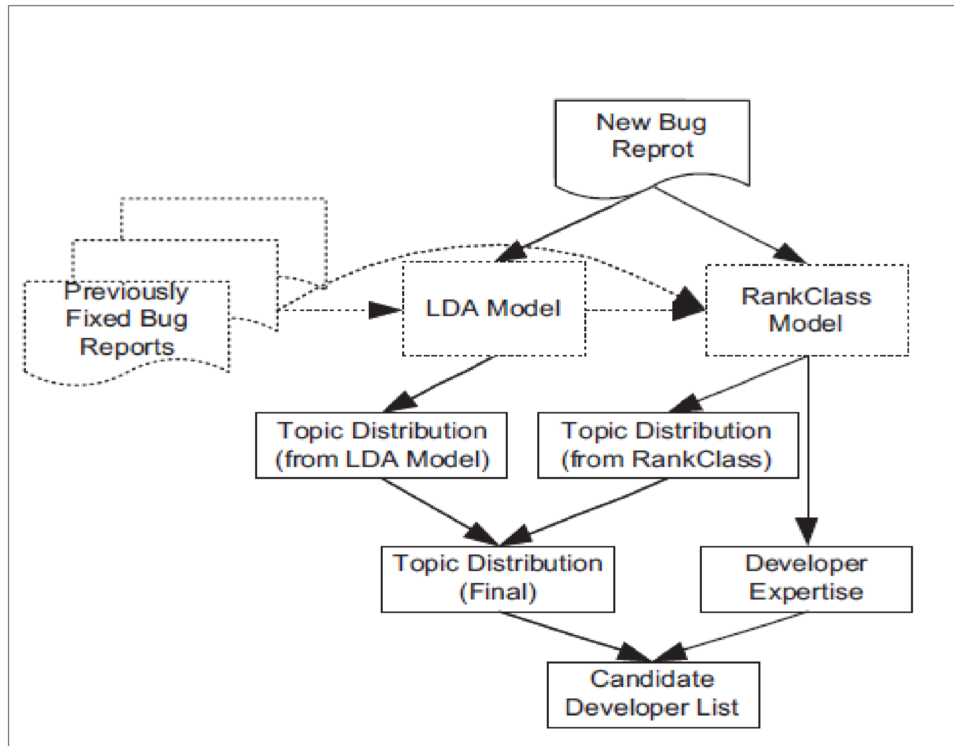


Figure 3.1: Overview of BUTTER model [48]

3.2.3 Developer Ranking Algorithm

Zhang et al. [49] presented a developer ranking algorithm for bug triage that utilizes topic models and developer relations. This approach involves several key steps. Bug reports are extracted and pre-processed, removing unnecessary information. Pre-processed data is input into the Topic Modeling Toolbox (TMT), which employs Latent Dirichlet Allocation (LDA) to generate topics and their associated terms. Candidate developers are derived from existing bug reports. The interaction between candidates and relevant topics is measured using metrics such as the number of assignments and comments. An "active reporter" is identified as the bug reporter with the most comments on their bug reports. Relations between candidates and the active reporter are calculated based on the number of fixed bug reports and related comments submitted by the diligent reporter. An algorithm is developed to rank candidate developers by combining Step 3 and Step 4 results. This approach is evaluated using three open-source projects (Eclipse, Mozilla Firefox, and NetBeans) and demonstrates superior performance in terms of F-score. However, it relies on historical bug fixing history and does not recommend new developers. Figure 3.2 shows the overall procedure of this approach.

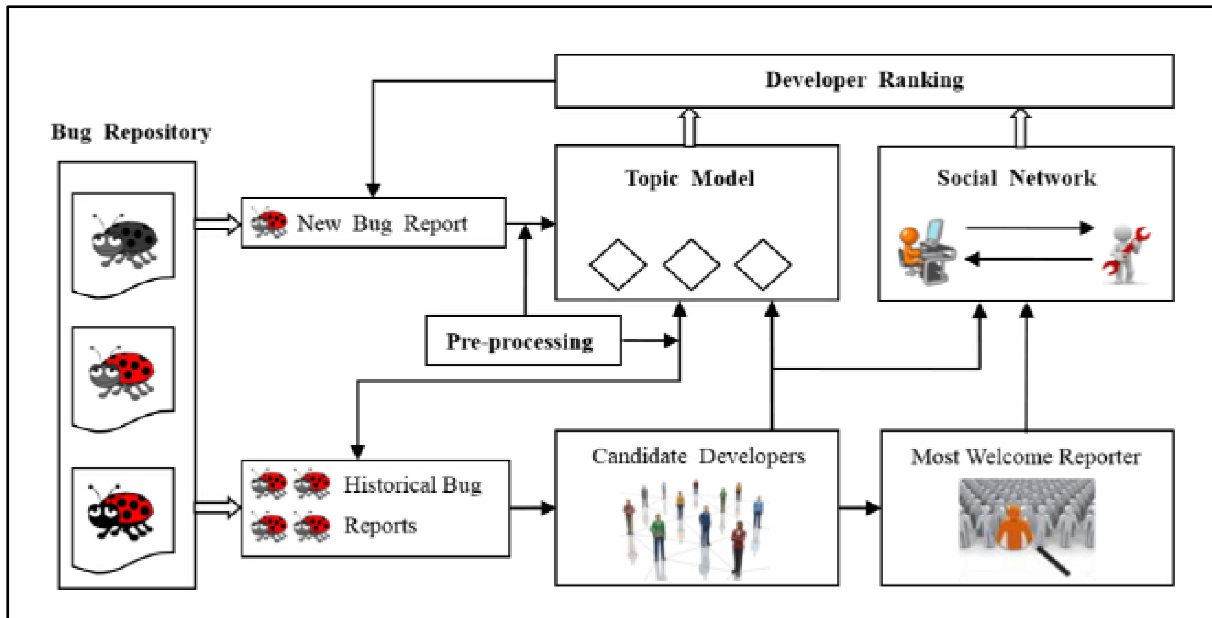


Figure 3.2: Overall framework of Developer Ranking Algorithm [49]

3.2.4 BAHA

Zhang et al. [50] introduced an automatic bug assignment technique called BAHA (Bug Assignment Approach with Topic Modeling and Heterogeneous Network Analysis). The bug triaging procedure of BAHA involves some steps. Textual contents (summaries and descriptions) are extracted from previous fixed bug reports and preprocessed. The LDA model is applied to the textual contents from existing bug reports to calculate topic distributions. A RankClass model is trained based on the outputs of the LDA topic models and textual contents from previous fixed bug reports. Upon the arrival of a new bug report, a heterogeneous network is created. The LDA model and the heterogeneous network are used together with the RankClass algorithm to estimate the topic distribution of the new bug report. Developers are selected based on the topic distribution and expertise scores. The top N developers are recommended for the bug. Tests conducted on the Eclipse JDT project demonstrate that BAHA outperforms other sophisticated automated bug assignment methods. However, like the previous approaches, BAHA relies on the history of bug fixing and does not recommend new developers. The proposed bug-triaging procedure of BAHA is shown in Figure 3.3.

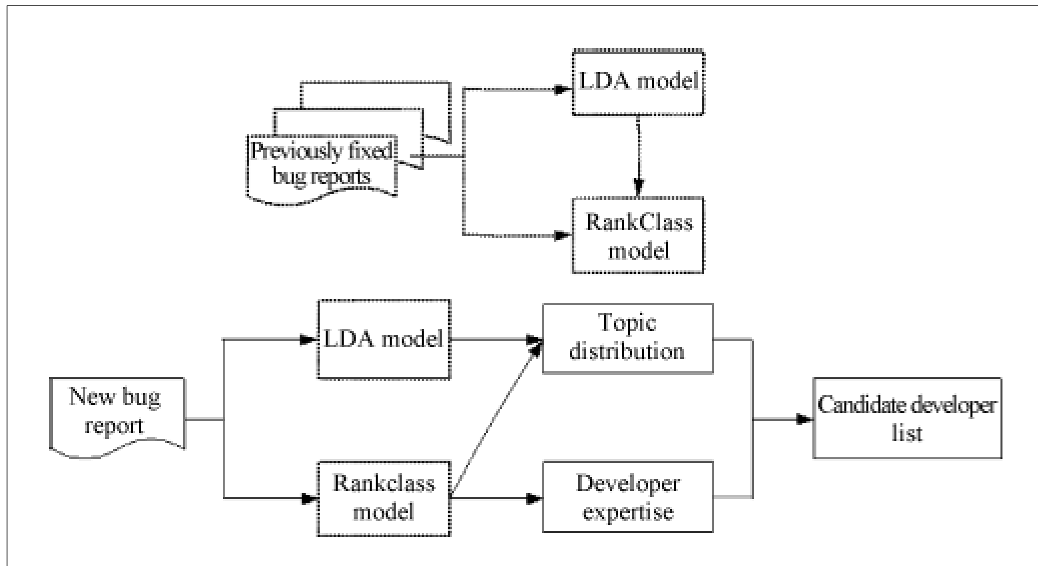


Figure 3.3: Overall framework of BAHA [50]

3.3 Information Retrieval-Based Approach

Information retrieval-based bug triage systems leverage historical data stored in software repositories. These repositories contain a wealth of information about the development and maintenance of software systems, including source code changes, commit messages, and version control records. Researchers use this data to make informed decisions about bug triaging. Information retrieval-based bug triage systems continue evolving, leveraging innovative techniques to recommend developers with the right expertise for resolving specific bug reports. These approaches draw upon the rich historical data stored in software repositories to streamline the bug-triaging process, ensuring that critical issues are addressed promptly and efficiently. Effective bug triage is essential for maintaining software quality and optimizing development resources, ultimately improving software reliability and user satisfaction.

3.3.1 Mining Software Repositories

Kagdi et al. [51] introduced a method that involves creating a corpus for each source code file using data from bug reports, such as descriptions, commits, source code snippets, and class or method identifiers. They employed latent semantic indexing to index this corpus, enabling them to quantify the similarity between bug report descriptions and the predicted source files. Developers were recommended based on their activities associated with these predicted files.

Shokripour et al. [52] proposed an automatic bug assignment approach that relied on extracting information from version control repositories. Their method utilized a phrase composition technique derived from commits and descriptions. This system recommended developers examine the activity histories in files with similar phrase compositions.

Additional studies [53] have explored noun extraction methods to identify bug locations by analyzing various information sources such as commit messages, comments, and source code. These methods calculate term-weighting schemes to predict the files relevant to a new bug report, and then developers are recommended based on their expertise with these predicted files.

3.3.2 Leveraging Latent Semantic Indexing

Latent semantic indexing is a popular information retrieval technique used in bug triage. It helps organize and understand textual data by uncovering the latent semantic structure in a large corpus of text documents. Linares-Vásquez et al. [54] adopted LSI by collecting identifiers, comments, and author information from source code files to create a corpus. This corpus was indexed using latent semantic indexing, and the similarity between files and bug report descriptions was computed. This approach recommended the authors of files with the highest similarity to the bug report.

3.3.3 Modeling Developer Expertise

Beyond textual data, some techniques focus on modeling developer expertise based on their interactions with various elements in the software development process. Naguib et al. [55] introduced a technique incorporating topic modeling and developer activities, including fixing, reviewing, and assigning bug reports. Developers are recommended based on their association scores within topics determined by their activities.

Yang et al. [56] presented approaches that utilize topic modeling and multiple features to identify candidate developers who have participated in bug reports with similar topics and features. These developers are then ranked based on their activities and contributions. S. Wang et al. [57] developed an unsupervised method that caches developers based on their activities at the component level. This approach calculates activeness scores for specific periods within the cache, assisting in recommending the most suitable developer for a bug report.

3.3.4 Enhanced LDA Methods

Some studies have explored the enhanced LDA methods to analyze the relationship between developers and bug reports. Xia et al. [58] extended the latent Dirichlet allocation (LDA) topic modeling algorithm by introducing a multi-feature LDA approach incorporating components and products. Within this enhanced paradigm, developers are suggested based on their affinity scores. Lee and Seo [59] devised a method to enhance triage performance by improving existing LDA topic sets. They introduced two adjunct topic sets constructed using multiple LDA-based topic sets, resulting in improved accuracy compared to conventional LDA-based methods.

3.3.5 Entropy-Based Optimization

Zhang et al. [60] introduced an entropy-based optimized LDA approach for building topic models for automatic bug report assignments. This approach utilized the Stanford topic modeling toolbox to train topic models using optimized LDA. Developer comments played a crucial role in modeling developer expertise and interest in specific topics, resulting in a ranked list of recommended developers.

3.3.6 Expertise Scoring and Ranking

Yadav et al. [61] proposed an approach that reduces bug tossing length and ranks developers based on their expertise in bug triaging. Developer profiles were generated based on their contribution performance, and expertise scores were calculated using various factors, including average fixing time, priority-weighted fixed issues, and index metrics. The approach considered feature-based, cosine, and Jaccard similarities to compute these expertise scores, ultimately providing a ranked list of developers for handling incoming bug reports.

Information retrieval-based bug triage systems continue evolving, leveraging innovative techniques to recommend developers with the right expertise for resolving specific bug reports. These approaches draw upon the rich historical data stored in software repositories to streamline the bug-triaging process, ensuring that critical issues are addressed promptly and efficiently. Effective bug triage is essential for maintaining software quality and optimizing development resources, ultimately improving reliability and user satisfaction.

3.4 Social Network Analysis-Based Approach

Social network analysis is one method that some researchers use to tackle the bug triage problem. In the software sector, developers work together closely to resolve bugs. The complex ties between developers and bug reports are considered by social network analysis approaches as a crucial component in identifying the best developer for a given task. Developers are viewed as nodes in this complicated problem model, and their collaborations are viewed as edges in a network.

3.4.1 Developer Social Network Construction

In the Developers Communities in Bug Assignment (DECOBA) approach by Banitaan and Alenezi, they focus on constructing a social network of developers [62]. This network is built based on the interactions and collaborations observed in the comments section of bug reports. By analyzing these interactions, DECOBA identifies groups or communities of developers who often work together or share expertise. These communities are valuable for recommending developers for new bug reports. The system ranks these developer communities to facilitate efficient bug assignment. This approach is beneficial for identifying patterns of collaboration within development teams.

3.4.2 Social Network Analysis with Machine Learning

In the approach that combines social network analysis with machine learning, as proposed by Zhang et al., the goal is to leverage both the social network characteristics of developers and their contributions to the bug resolution process [63]. Metrics such as the number of bug fixes, comments, and reports are considered to calculate a developer's contribution score. This score is integrated with a classifier score derived from machine learning algorithms. By combining these two scores, the approach determines the most suitable developer for a given bug report. This hybrid approach enhances bug assignment accuracy by considering social interactions and individual developer performance.

3.4.3 Information Retrieval for Bug Similarity

Zhang and Lee's approach revolves around information retrieval techniques to find bugs similar to the triaged ones [64]. By identifying similar bugs, the system can recommend a developer with experience or expertise in addressing issues of a similar nature. The fixing

probability, which indicates the likelihood of a developer successfully resolving a particular bug, is determined using social network analysis. The fixing experience score is also calculated based on the developer's history of fixing and assigning bugs. Combining these factors, the approach provides a recommendation for bug assignment that is informed by the developer's social network connections and experience in handling similar issues.

3.4.4 Bug-Fixing Expertise and Association-Based

Hu et al.'s bug-fixing technique focuses on assessing the expertise of developers in bug resolution. It calculates the similarity between the target bug and other bug reports in the repository. Developers are recommended based on their associations with specific components and bugs. This approach emphasizes the developer's familiarity with the details and issues of the bug being triaged. Considering these associations, the system optimizes the bug assignment process by suggesting well-suited developers to address the particular problem, thereby increasing the chances of efficient bug resolution [46].

3.4.5 Concept Profile and Social Network

In their study, Zhang et al. [65] introduced a bug triage process that combines concept profiles and developer social networks to rank suitable developers based on their bug-solving experience and cost-effectiveness. This method involves three main steps:

i) Concept Profile Creation

To establish concept profiles, the authors undertake two procedures. They utilize the K-means clustering algorithm to categorize bug reports in the training dataset. Subsequently, they gauge the textual similarity between bug reports by employing the cosine measure, which considers both the title and description of bug reports. Bug concepts are identified after clustering existing bug reports in the training dataset. They extract topic terms frequently appearing in bug reports associated with the same concept. Figure 3.4 shows an example of a topic term. Normalization is applied to convert term frequency into weight values. Determining topic terms involves setting a threshold value ($\theta 1$), which is surpassed by the weight value of terms.

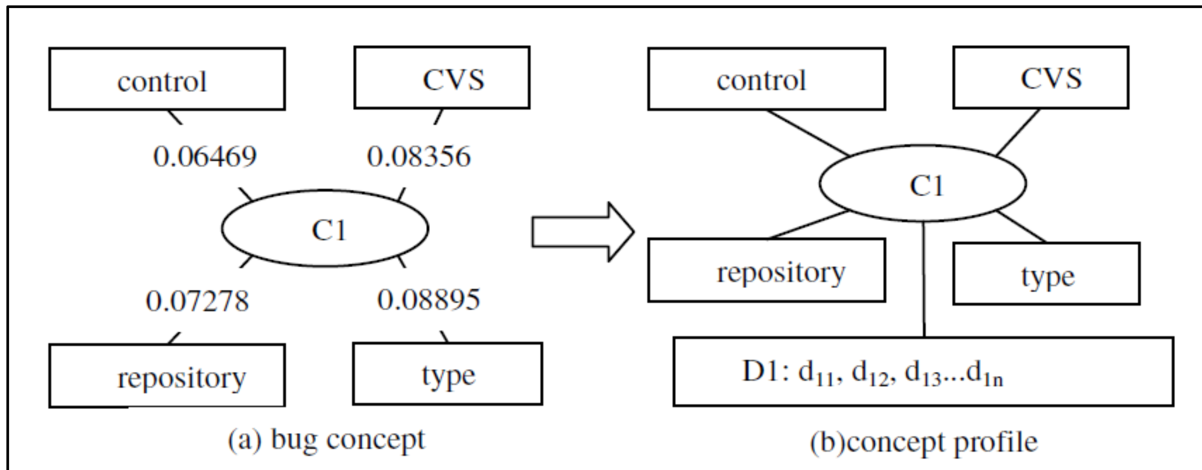


Figure 3.4: Bug concept with the topic terms, where θ=0.05 [65]

ii) Retrieving Candidate Developers Using Social Network

When a new bug report emerges, it assesses its relevance to existing bug concepts. In a social network with five nodes (Figure 3.5), the "concept" node represents a group of developers derived from the concept profiles, with links indicating their relationships.

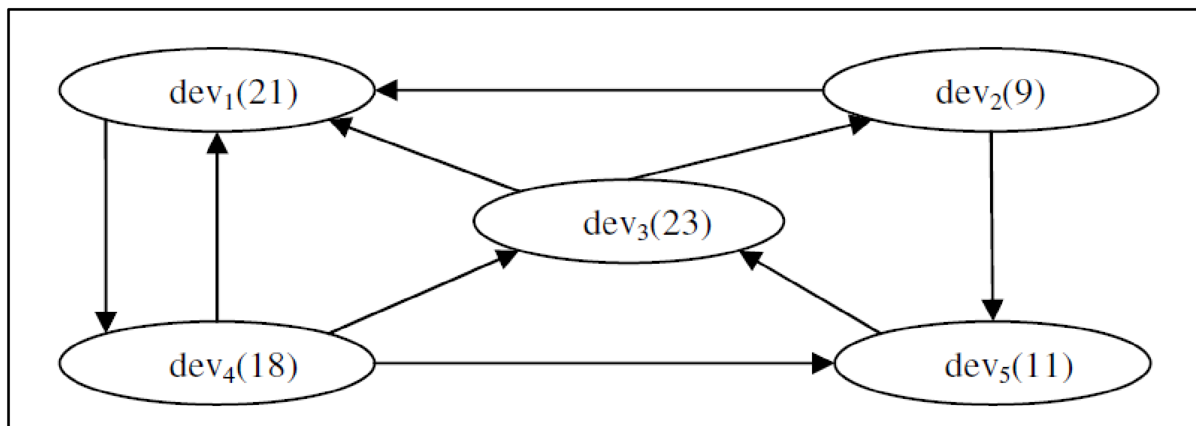


Figure 3.5: An example of social network [65]

iii) Ranking Candidate Developers

The authors develop a ranking algorithm to select the most suitable developers from a list of candidates for fixing a given bug. This algorithm recommends that the top k developers address the new bug report. It's essential to note that this method relies solely on previous bug-fixing records and does not provide recommendations for newly-appointed developers.

These sub-sections showcase various social network analysis-based bug triage systems aspects, including network construction, machine learning integration, information retrieval, and expertise assessment. Researchers and practitioners can explore these approaches to tailor bug-triaging strategies to their needs and development environments.

3.5 Dependency Based Approach

Dependency-based bug Triage Systems are a bug triaging approach that considers the dependencies between software components when assigning bug reports to developers. These systems aim to improve the accuracy and efficiency of bug assignment by considering the relationships and dependencies among different parts of a software project.

In traditional bug triage systems, bug reports are often assigned based on factors such as developer expertise or historical bug-fixing patterns. However, these systems may need to consider the interdependencies between software components and modules. Dependency-Based Bug Triage Systems fill this gap by analyzing the structure and identifying which components are affected by a bug.

3.5.1 Bug Dependency-Based Mathematical Model

Kumari et al. [66] tackled the bug triage problem by introducing a bug dependency-based mathematical model. This approach recognizes that bugs often depend on coding errors, architectural flaws, or misunderstandings between users and developers. To determine developer assignments, the model utilizes information from the bug report, such as the summary, description, and comments, to calculate entropy. Developers are assigned based on the entropy measurement. By considering bug dependencies, this approach aims to optimize bug assignment by matching developers' expertise with the nature of the bug's dependencies.

3.5.2 Scheduling-Driven Task Assignment

Etemadi et al. [67] proposed a scheduling-driven approach for efficient bug-fixing task assignments. Their method employs a task dependency graph where each task corresponds to a node. Tasks are associated with both a starting time and an ending time. The approach utilizes an embedded greedy search algorithm that operates on schedules to effectively explore different parts of the search space. This strategy enhances accuracy in bug assignment and reduces the time required for bug resolution. By scheduling tasks and managing dependencies, developers can work on bugs more efficiently.

3.5.3 Automated Bug Triage with Dependency

Almhana and Kessentini [68] introduced an automated bug triage method that considers dependencies among bug reports. Their approach involves localizing the specific files that

need inspection for each open bug report. It employs multi objective search techniques to rank bug reports for programmers based on dependencies with other reports and their associated priorities. This method demonstrates significant time savings, with over a 30% reduction in the time required for localizing bugs concurrently compared to traditional bug prioritization techniques. This approach streamlines the bug triage process by addressing bug dependencies and optimizes developer efforts.

3.5.4 Dependency-Aware with NLP and Integer Programming

Jahanshahi et al. [69] proposed a dependency-aware bug triage method that combines natural language processing and integer programming. Unlike previous dependency-based approaches, this method integrates textual information, the relationships between bugs, and the cost associated with each bug. By considering these factors, it aims to assign bugs to developers to minimize overdue bug reports and improve bug-fixing time. However, it should be noted that this approach assumes each developer can work on only one report at a time, which may only sometimes align with the practical realities of development processes.

These sub-sections highlight various aspects of dependency-based bug triage systems, encompassing mathematical modeling, scheduling-driven approaches, automation with dependency consideration, and integrating NLP and integer programming. These techniques offer valuable insights into optimizing bug assignment by considering dependencies and associated factors. Researchers and practitioners can explore these approaches to enhance their bug-triaging strategies in real-world software development scenarios.

3.6 Machine Learning-Based Approach

Over the past decade, many bug triage methods based on machine and deep learning have been introduced. These methods treat bug reports as learning instances and frame bug triage as a classification problem. To categorize these approaches, we classify them into two main groups:

- Conventional machine learning-based bug triage systems
- Deep learning-based bug triage systems

This classification helps effectively organize the diverse range of techniques used to address bug triage challenges.

3.6.1 Conventional Machine Learning-Based

Conventional Machine Learning-Based Bug Triage Systems rely on various feature extraction methods, including TF-IDF, chi-square for identifying discriminative terms, term selection, and mutual information. These techniques are critical in transforming textual bug report data into numerical features that machine learning algorithms can work with effectively.

3.6.1.1 Bug Triage Using Selected Fields

Researchers like Bhattacharya and Neamtiu [70] focus on specific fields within bug reports, such as titles, summaries, descriptions, and additional attributes. They extract relevant features from these fields using techniques like TF-IDF and the bag-of-words (BOW) model. Subsequently, machine learning classifiers, such as Naïve Bayes, assign the most suitable developer to the reported bug.

3.6.1.2 Fuzzy Set Features for Bug Triage

Tamrawi et al. [71] introduce a distinctive approach to bug triage by incorporating fuzzy set features. This technique analyzes bug report titles and descriptions and extracts critical terms as features. Fuzzy set features provide a way to represent the imprecise or uncertain nature of bug descriptions, contributing to more nuanced bug assignments.

3.6.1.3 Generalized Recommendations for Developers

Anvik and Murphy [72] expand their bug triage strategy beyond individual bug fixer assignments. They employ normalized TF-IDF for feature extraction from titles and descriptions and experiment with various machine learning algorithms, including Naïve Bayes, expectation-maximization, SVM, Decision Trees (C4.5), K-Nearest Neighbor, and conjunctive rules. Their approach goes beyond assigning a single bug fixer by recommending components and suitable developers.

3.6.1.4 Developer Prioritization with TF-IDF

Xuan et al. [73] prioritize developers in their bug triage process using TF-IDF for feature extraction from titles and descriptions. This prioritization helps ensure that the most appropriate developer is assigned to address a particular bug. Naïve Bayes and SVMs are leveraged as classifiers to facilitate bug assignment.

3.6.1.5 Bug Triage with Metadata Consideration

Banitaan and Alenezi [74] take bug report metadata into account to enhance triage accuracy. They utilize feature extraction methods like TF-IDF and chi-squared techniques. The Naïve Bayes classifier is employed for the bug assignment task. This approach aims to improve prediction accuracy by considering additional information associated with bug reports.

3.6.1.6 Automatic Developer Assignment with Discriminatory Terms

Alenezi et al. [75] introduce an automated approach for assigning developers with relevant experience to new bug reports. Their method employs a five-term selection method, which includes the chi-square method, log odds ratio, term frequency relevance frequency, mutual information, and distinguishing feature selector. The Naïve Bayes classifier is used to select bug fixers for new bug reports, with the chi-squared term selection method proving particularly effective.

3.6.1.7 Data Reduction for Improved Accuracy

Xuan et al. [76] employ a data reduction technique to select instances and features, contributing to higher accuracy in bug triage. This technique, in conjunction with the Naïve Bayes classifier, helps enhance the accuracy of bug assignment by reducing noise and irrelevant information.

3.6.1.8 Ensemble Classifier for Enhanced Bug Triage

Jonsson et al. [77] employ ensemble learning techniques to improve bug triage results. They extract features using TF-IDF from titles and descriptions and create a stacked generalizer classifier. This classifier combines various base classifiers, including the Bayes net, Naïve Bayes, SVM, KNN, and decision tree classifiers. Ensemble learning enhances the robustness and overall performance of the bug triage system.

3.6.1.9 SVM-Based Bug Recommender System

Florea et al. [78] propose a bug recommender system based on Support Vector Machines and feature extraction techniques like TF-IDF and chi-squared. Their model is tested on multiple datasets and focuses on preserving nouns from text attributes (such as summaries and descriptions). The use of SVM enhances the accuracy of bug recommendations.

3.6.1.10 Incorporating Categorical Features and Metadata

Alenezi et al. [79] explore the integration of categorical features and metadata alongside textual data for bug triage. They utilize the gain ratio to identify essential features, specifically emphasizing operating systems and priority determination from metadata. Combining textual data with categorical features shows promise in improving bug triage outcomes.

3.6.1.11 High-Confidence Bug Triage

Sarkar et al. [80] introduce a bug triage system incorporating high-confidence prediction levels. They utilize alarms and crash dumps in addition to textual and categorical attributes. Feature extraction methods like normalized TF-IDF and line-IDF are applied to textual data and alarm/crash dumps. Classification tasks are performed using logistic regression, SVM, KNN, and Naïve Bayes classifiers, with logistic.

3.6.1.12 Semi-Automated with Skill-Based Developer Recommendation

Anvik et al. [81] introduced a semi-automated bug triaging approach designed to proficiently assign newly reported bugs to developers based on their relevant skills. This method employs a supervised machine learning algorithm to propose a concise list of developers frequently engaging with similar issues. The process combines automation with human expertise, as a trigger ultimately selects the most appropriate developer from the suggested set. The methodology begins by characterizing bug reports, extracting pertinent features, and grouping them based on similarity, primarily focusing on the summary and description of each report. The text is transformed into a feature vector to facilitate the application of machine learning algorithms to the free-form text in these reports. Standard preprocessing techniques are initially applied, including removing stop words and non-alphabetic tokens. The resulting words are then used to construct a feature vector that captures the term frequencies. Next, each bug report is labeled with the developer's name who resolved it, providing valuable training data. To enhance the dataset's quality, filters are employed to refine bug reports that lack project-specific, useful labels. Additionally, developers who are no longer active on projects or have resolved only a limited number of bugs are excluded from consideration. The bug assignment process utilizes Support Vector Machines to propose a list of potential developers, which is subsequently presented to a human trigger for final recommendation. The human trigger selects the most suitable developers from this list based on their expertise and availability. This approach was evaluated on two prominent open-source projects,

Eclipse and Firefox, achieving a precision rate of 57% for Eclipse and 64% for Firefox. The semi-automated bug triaging method effectively combines machine learning capabilities with human judgment to streamline the bug assignment process and enhance accuracy.

3.6.1.13 Enhanced Bug Triage through Integrated Models

Zhang et al. [64] introduced an integrated bug triage algorithm that combines probability and experience models to enhance developer recommendation. Their approach involves preprocessing new bug reports, retrieving similar historical bugs, extracting relevant features, and considering factors like the fixer's name, the time required for fixes, and the number of re-opened bugs. Candidate developers are identified, and a probability model analyzes their potential for resolving new bugs. Simultaneously, the experience model evaluates the historical performance of developers. These models are merged into a hybrid bug triage algorithm to recommend suitable developers. Evaluation of open-source projects JBoss and Eclipse demonstrates the effectiveness of this method in recommending appropriate developers for bug resolution. Figure 3.6 represents the probability and experience models used in this algorithm.

3.6.1.14 Activity-Based Bug Triage Strategy

Naguib et al. [55] proposed an activity-based bug-triaging strategy that creates activity profiles for bug-tracking system developers. This approach leverages Latent Dirichlet Allocation (LDA) to establish topic models for bug reports, categorizing words into topics from report titles, descriptions, and system components. An activity profile is generated for each developer using historical logs, encompassing their roles and topic associations. When a new bug report arrives, it extracts its topic model and ranks developers based on their activity profiles and topic relevance. This strategy achieves an average hit ratio of 88% across three different projects, outperforming the LDA-SVM-based assignment recommendation technique.

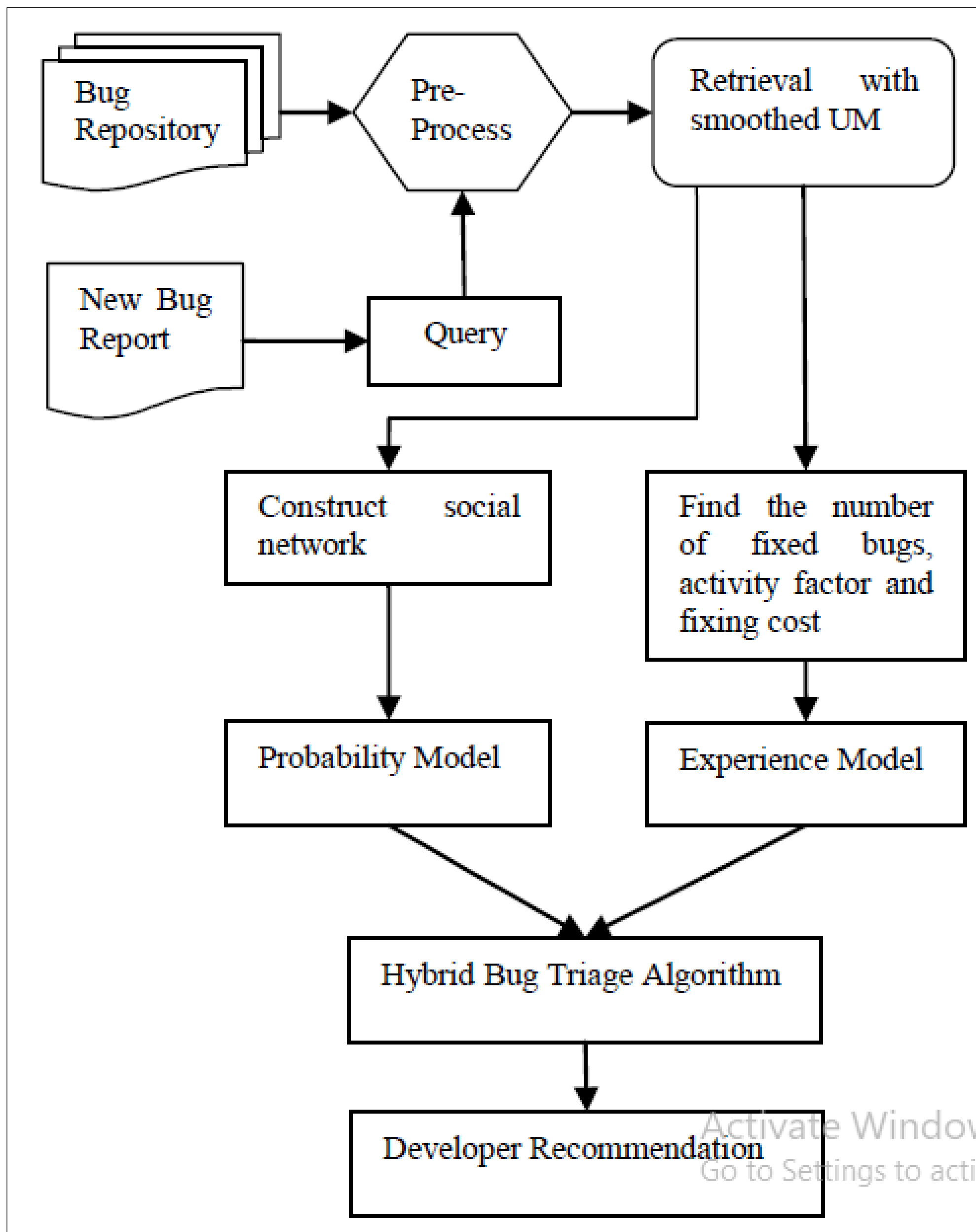


Figure 3.6: Structure of hybrid bug triaging [55]

3.6.1.15 BugFixer

Hu et al. [46] introduced BugFixer, a bug-triaging method that capitalizes on past bug-fixing information. BugFixer creates a Developer-Component-Bug network, connecting developers to components and components to bugs. The methodology involves two main parts: Bug Report Similarity and The DCB Network. Bug Report Similarity employs a unique tokenization algorithm to enhance bug report similarity calculations, using SVM to assess matches. The DCB Network leverages historical bug-fix data to determine relationships between new bug reports and developers. BugFixer excels in larger projects and performs

comparably well in smaller projects, as evidenced by evaluations on open-source and industrial projects. Figure 3.7 shows the overall structure of BugFixer.

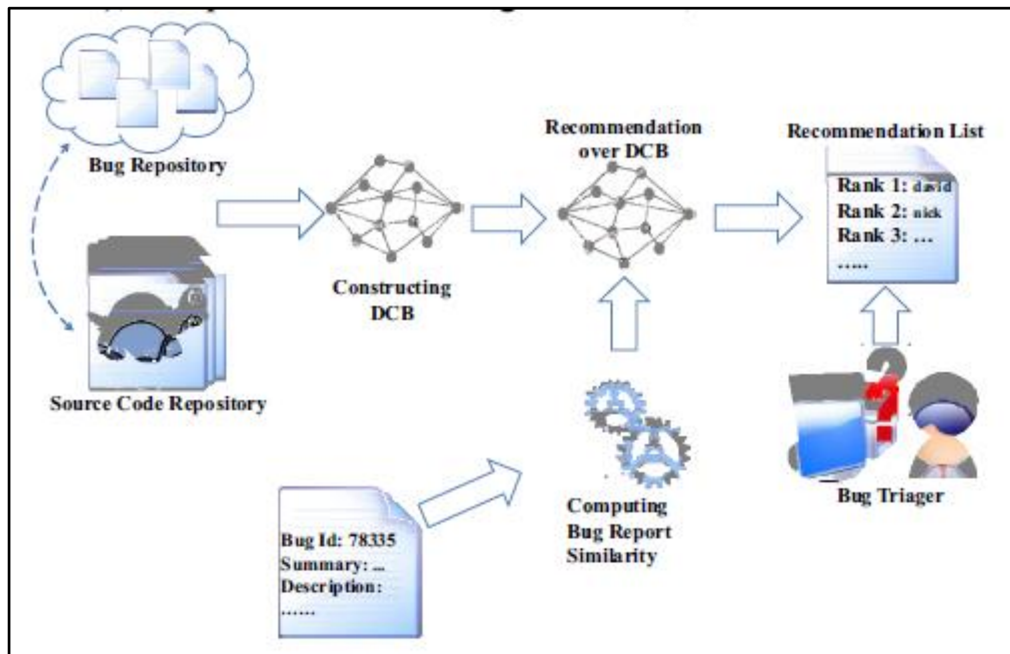


Figure 3.7: Overall structure of BugFixer

3.6.1.16 Search-Based Bug Triage with Apache Lucene

Peng et al. [82] propose a search-based bug triaging method using Apache Lucene to identify relevant bug reports and corresponding developers. The approach involves creating an index for bug reports and crafting queries based on product, component, summary, and description from bug reports. The search results yield ranked developers, with the top N recommended for resolving the bug. However, this method focuses exclusively on existing experienced developers and does not incorporate new developers into the recommendation process.

3.6.1.17 Common Vocabulary-Based Bug Triage Algorithm

Nagwani et al. [83] introduced a bug triaging algorithm that identifies relevant developers for new bug reports by creating common words set from existing bug reports. It compiles a developer list from the bug archive, generates developer vocabulary lists from previous fixed bug reports, and maps them with new bug report common words. This process generates a list of developers for solving new bug reports. However, the algorithm primarily relies on known developers and does not recommend new developers for bug resolution.

3.6.2 Deep Learning-Based

Recently, bug triage has witnessed significant advancements with the adoption of Natural Language Processing and deep learning techniques. These cutting-edge approaches leverage the power of NLP to handle word embedding and word representation, presenting innovative solutions to bug triage challenges that have emerged over the past few years.

3.6.2.1 CNN-Based Bug Triage with Word2Vec

S. Lee et al. [84] introduced a bug triage approach that leverages Convolutional Neural Networks and the word2vec model for word representation. This model utilizes bug report summaries and descriptions as input data. They validated this technique across two open-source and industrial projects, marking a significant step in adopting CNN for bug triage. Before this, CNN was primarily employed for various other software engineering tasks, including bug detection, severity classification, and identifying code smells.

3.6.2.2 Multilabel Deep Neural Network for Bug Triage

Choquette-Choo et al. [85] proposed a multilabel and dual-output deep neural network for bug triage systems. The authors employed latent semantic analysis and introduced a two-output deep neural network architecture to achieve latent space representation. This architecture initially predicts team classes and developers based on the predicted team. Their approach incorporates a heuristic process that considers bug fixer information and developer contribution levels, providing a comprehensive method for bug triage.

3.6.2.3 Deep Bidirectional RNN with Attention for Bug Triage

An attention mechanism-equipped deep bidirectional Recurrent Neural Network (DBRNN-A) was introduced by Mani et al. [86] to learn synthetic and semantic characteristics from bug report summaries and descriptions. They utilized a representation based on DBRNN-A for classifier training and vectorizing textual data using word2vec. The authors also shared their data, creating benchmark datasets for future research. This approach enhances bug triage with the ability to capture contextual information and semantic relationships.

3.6.2.4 Activity-Based Bug Triage with CNN

S. Guo et al. [87] proposed an activity-based bug triage technique that utilizes the CNN model. Their approach involves sorting data based on the bug report creation time, using the

last 10% of the data for testing. The technique demonstrated promising results, mainly when applied to large datasets. This method considers the temporal aspect of bug reports and leverages CNN for effective triage.

3.6.2.5 CNN-Based Bug Fixer Recommendation System

Zaidi et al. [88] introduced a CNN-based bug fixer recommendation system that utilizes both small and large datasets. They employed various word embedding techniques, including word2vec, GloVe, and ELMo. Multiple convolutional kernels were used to extract diverse features from bug reports. This approach showcased state-of-the-art performance in bug triage, emphasizing the importance of word embeddings and convolutional neural networks in recommendation systems.

3.6.2.6 Heterogeneous Graph-Based Bug Triage

A recent development in bug triage involves a heterogeneous graph-based method utilizing Graph Convolutional Networks (GCN). This approach generates heterogeneous graphs from triage history data, offering a quicker alternative to CNN and RNN methods while delivering comparable results. Heterogeneous graphs encompass various relationships between data entities, providing valuable context for bug triage [89].

3.6.2.7 Multitriage Model for Developer Assignment

Aung et al. [90] proposed a multitriage model capable of assigning developers and issue types simultaneously. This innovative approach employs two different deep-learning models for feature extraction. The text encoder module is based on the CNN model, while the abstract syntax tree encoder module utilizes biLSTM. Features from both encoders are concatenated, and two separate classifiers are trained—one for developer assignment and another for bug issue type classification. While effective, this model requires additional training time due to its dual-encoder and dual-model architecture.

3.6.2.8. Bug Triage with Graph Neural Network

Zaidi et al. [2] presented a bug triage system utilizing a Graph Neural Network (GCN) and a heterogeneous graph. This approach constructs a heterogeneous graph that includes word-word and word-bug document edges, with TF-IDF used for weighting word-bug document edges. Different similarity metrics are employed to weight word-word edges. A simple two-layer GCN trains the model, recommending ten developers for a given unseen bug report.

This method capitalizes on the power of graph-based representations and neural networks for efficient bug triage.

These advanced bug triage methods, incorporating NLP and deep learning techniques, offer improved accuracy, semantic understanding of textual data, and the ability to handle complex relationships, ultimately enhancing the bug triage process in software development. Researchers continue to explore these methods to optimize bug assignment and developer recommendations further.

3.7 Reassignment-Based Approaches

In the context of bug triage, reassignment occurs when the initially assigned developer is either unable or unwilling to resolve a bug, prompting the transfer of the bug to another developer for resolution. This process of bug reassignment has notable drawbacks for the overall maintenance process. The primary reason behind bug reassignment lies in the manual assignment of tasks, which can be error-prone. Consequently, bug reassignment introduces additional costs and prolongs the time required to resolve bugs. Studies in this area have revealed that a significant portion of bugs, ranging from 37% to 44%, undergo at least one reassignment event. Furthermore, the average duration of a single reassignment event is approximately 50 days. As such, reducing these instances of bug reassignment can enhance the efficiency and effectiveness of bug-triaging systems.

3.7.1 Tossing Graph-Based Approaches

In the context of bug reassignment, researchers have employed tossing graphs, which are essentially directed graphs where nodes represent system developers and directed edges symbolize the transfer of a bug from one developer to another. These graphs visually represent the bug passing history, where shorter paths indicate fewer reassignments. Several bug triaging techniques have been developed based on these tossing graphs to minimize the number of bug reassignments caused by incorrect developer assignments.

3.7.1.1 Bug Tossing Graphs

Jeong et al. [91] introduced bug-tossing graphs to enhance bug assignment techniques. These graphs represent bug passing history among developers. Two models are employed: the actual model and the goal-oriented model. The actual model captures all tosses, while the goal-oriented model simplifies it. Inspired by Markov Chain properties, weighted edges

represent the probability of a toss between developers. When a new bug report arrives, machine learning identifies similar bug reports, and developers with tossing solid relationships are recommended. The approach improves prediction accuracy but may face search failures.

3.7.1.2 Enhanced Tossing Graphs

Building on the previous work, Chen et al. optimized tossing graphs by pruning retired or non-recent developers [92]. When a new bug report arrives, the Vector Space Model identifies similar previous bugs, and the tossing graph is pruned based on these similarities. Developers in the pruned sub-graph are recommended as potential developers. Evaluation metrics, like Mean Length of Tossing Paths (MLTP) and Failure Rate (FR), indicate successful bug fixer identification with fewer tossing events. These tossing graph-based approaches tackle bug reassignment challenges by leveraging historical tossing patterns to enhance developer recommendations.

3.7.2 Multi-Feature Incremental Learning

Bhattacharya and the team [93] proposed an advanced approach that enhances previous techniques by incorporating multiple bug report features such as product and component during graph construction. A significant contribution of their work is the introduction of incremental learning, performed after recommending each new bug report. This incremental learning approach continually updates the training set with each new instance, improving the model's learning capability over time. Traditional tossing graphs are built solely from past bug reports, which can limit recommendation accuracy as they rely exclusively on historical data. One notable limitation of these approaches is their struggle to accommodate new developers effectively. Since new developers lack bug-fixing histories, they need to be integrated into the graph, making it challenging to ensure equal resource utilization and team recommendation, which are vital in industrial settings. Bhattacharya, et al.'s method, addresses these limitations by considering various bug report features and implementing incremental learning to adapt and improve developer recommendations as new bug reports emerge.

3.8 Text Categorization-Based Approaches

Text categorization-based approaches in bug triage involve the development of models trained on historical bug reports. These models utilize text similarity algorithms to compare new bug reports with the knowledge gained from past reports. The primary objective of such approaches is to predict the most suitable developers by ranking them based on their previous experience in resolving similar bugs. This methodology aims to recommend developers with relevant expertise for efficient bug resolution.

3.8.1 Bug Report Meta Data

Bug report metadata is essential for bug assignment techniques, where bug reports are treated as instances, textual descriptions as features, and developers as labels. However, noisy and irrelevant terms in the textual content can hinder accuracy. Banitaan et al. [74] introduced TRAM (Term-based Representation with Assignment Metadata) to overcome this. To improve accuracy, TRAM utilizes metadata, including discriminating terms, bug components, and reporters. It extracts features from the bug report metadata, like the title. Preprocessing techniques are applied, and a weighted bug-term matrix is constructed. Discriminating terms are selected using the Chi-Square method. The predictive model is built using the Naive Bayes Classifier for its efficiency and accuracy. TRAM was tested on open-source projects, achieving higher precision and recall than baseline approaches. However, it has limitations, such as assuming a single expert developer for each report and not considering recent developer activity, which can affect the assignment of new developers.

3.8.2 Developer Preference Elicitation

Developer preference plays a crucial role in efficient task assignment, as it taps into a developer's expertise and aligns with their motivation. Research on the Eclipse project indicates that a significant portion (24%) of bugs is reassigned to different developers before resolution, underscoring the importance of considering developer preferences in the assignment process. In addressing this concern, Baysal et al. [94] introduced a preference elimination technique to enhance existing text categorization methods for bug assignment. Their framework comprises three core components. The approach begins with the assumption that developers are inclined to tackle bugs within their domain of expertise. The Expertise Recommendation component employs the Vector Space Model to deduce developer expertise

from their prior bug fixes. To build a developer profile, it extracts keywords from various textual sources, such as bug summaries, descriptions, and comments. The term vector is weighted using the tf-Idf scheme to emphasize relevant terms. When a new bug report surfaces, a comparison is made between the term vectors of developers and the incoming report to recommend developers with relevant expertise. This component allows developers to express their willingness to address specific bugs. A "Rating" field is introduced within bug reports to collect and store preference levels through feedback. Developers categorize bugs as "Preferred," "Neutral," or "Non-Preferred." The developer's preference is determined by creating a whitelist of bugs they rated as preferred. This whitelist consists of a term vector comprising terms from the preferred bug reports. When a new bug emerges, its similarity to these whitelists is assessed to assign preference levels to developers. The Task Allocation component combines developer preference and expertise to suggest suitable fixers. The technique also factors developers' current workload and availability using a method proposed by Weiss et al. However, the study needed to provide experimental validation for this approach due to the complexity of model design and the unavailability of developer details. Furthermore, it acknowledges potential limitations, such as the possibility of developers manipulating their preference ratings and the challenge of new developers receiving no tasks as they need a bug-fixing history.

3.8.3 Code Authorship

Traditional bug triage approaches often require extensive mining of software repositories, typically focusing on either bug reports or source code commit histories. These processes involve significant computational costs and resource-intensive data collection efforts. To address these challenges, Hossen et al. [95] introduced an innovative triaging approach centered on code authorship for incoming change requests, whether they involve bug fixes or new features. This approach comprises two key steps. Initially, the project's source code under consideration is indexed using Latent Semantic Indexing. This indexing process specifically extracts identifiers and comments from the source files. Consequently, each source file is associated with a vector within the constructed index. When a new change request arrives, its lengthy description is extracted and matched against the indexed files using LSI. This results in a ranked list of relevant source files. The subsequent step involves recommending a list of developers based on the identified source files. The source code files are transformed into a lightweight XML representation called "srcML." Header comments,

typically containing copyright, licensing, and authorship information, are extracted from this XML representation. To obtain author information, regular expressions are devised and applied to the textual header comments. The relevance of an author is determined by calculating the frequency of their name appearing in the comments of the top-ranked source files. In cases of tiebreakers, the rank of these source files is considered. The effectiveness of this approach was demonstrated through its application to three open-source projects, where it outperformed two benchmark systems in terms of precision and recall. However, the approach's validity is contingent on the quality of comments provided by authors, which can be inconsistent and may not adhere to standard coding conventions. Additionally, authors may choose not to disclose their information in source file comments. Moreover, the method needs to include consideration of developers' expertise, potentially leading to assignments to inexperienced developers. Developers who have to comment on any source files may receive no tasks and miss out on future change requests.

3.9 Bug Data Reduction Approaches

In bug triaging, some approaches that focus on data reduction to enhance effectiveness have emerged [96]. These methods eliminate noisy and duplicate data from a system's bug repository. However, data reduction approaches have their limitations. The reduced dataset represents historical developer activity-based information, which can lead to reduced prediction accuracy when dealing with inactive or retired developers. Furthermore, these reduced datasets need to be updated when new developers join, resulting in new developers being excluded from task assignments.

3.9.1 Source-Based Bug Assignment Approaches

In the domain of bug assignment, researchers have also explored source-based techniques. These approaches are rooted in the idea that having an externalized model of a developer's expertise, particularly in code commits, can enhance task assignment. Building on this concept, developer vocabulary-based methods have been developed to capitalize on developers' source code activities. These approaches prioritize leveraging developers' source code contributions and expertise in the bug assignment process. One notable approach, proposed by Xuan and colleagues [76], combines various data reduction techniques, including instance selection (e.g., removing duplicate bug reports) and feature selection (e.g., identifying essential keywords). By applying feature and instance selection algorithms to an

existing bug repository, this technique reduces both the number of bug reports and the word dimension within those reports. The resulting reduced bug data contains fewer bug reports and a smaller vocabulary than the original dataset while conveying similar information.

3.9.2 Developer Vocabulary-Based Approach

To suggest developers for bug assignment, Matter et al. presented Develect, a vocabulary-based expertise model [97]. This approach starts by parsing the source code and constructing a model of the entire codebase. It employs the 'diff' command to capture word frequencies in changes made between two versions of the same file. The words found in identifier names and comments within the altered files are considered part of a developer's vocabulary. An expertise model is then generated using existing vocabularies and stored in a matrix format known as a term-author matrix. Develect compares the keywords in the reports with developer vocabularies when new bug reports arrive, using lexical similarities. Developers with the highest scores are identified as potential bug fixers. A threshold value is applied to address the issue of recommending inactive developers. However, this approach disregards experienced developers in the recommendation process, potentially leading to the suggestion of novice or inexperienced developers. Additionally, source code comments can introduce noisy information, as developers might include irrelevant comments that affect the accuracy of the approach. Furthermore, Develect struggles to model new developers who have yet to make any code commits, which can result in increased bug reassignments, longer resolution times, and reduced recommendation accuracy.

3.9.3 Commit Time-Based Approach

Many existing bug assignment approaches overlook the temporal aspect of developer commits, which can be a crucial indicator of their current activities. Considering the time metadata associated with code commits can help reduce the recommendation of inactive developers. One approach that incorporates time metadata into bug assignment is the ABA-Time-tf-idf technique [98]. This method identifies recent developers for bug assignments using time metadata as a key factor. It begins by parsing various source code entities, such as class names, method names, method parameters, and class attributes, and associates these entities with contributors to build a corpus. When new bug reports arrive, the technique searches for keywords in the index and assigns weights based on their usage frequency and associated time metadata. This means that developers who have used specific terms more

recently will receive greater emphasis in the recommendation process and will be listed at the top of the recommended developers' list. However, ABA-Time-tf-idf lacks high accuracy because it overlooks experienced developers and does not adequately support industrial requirements, such as team recommendation and resource allocation, especially in the context of new developers.

3.10 Cost Aware Based Approaches

Park and colleagues [99] introduced CosTriage, a developer ranking algorithm incorporating cost considerations into the bug triage process. This approach transforms bug triaging into an optimization problem, balancing accuracy and cost-effectiveness. It utilizes Content Boosted Collaborative Filtering (CBCF) to rank developers, addressing the questions of "Who can fix the bug?" and "Who can fix it faster or at a lower cost?"

CosTriage consists of two main steps: constructing developer profiles and ranking developers. Developer profiles are numeric vectors representing estimated costs for developers to fix specific bugs. To determine bug types, Latent Dirichlet Allocation (LDA) is applied to the bug repository. After identifying bug types, the approach computes the average fix time for each bug type per developer to create developer profiles. If a developer lacks a history of fixing a particular bug type, their profile contains missing values, which are filled in using collaborative filtering. When a new bug report arrives, CosTriage first identifies the bug type. It does so by extracting words from the report's title and description, then calculating the word distribution for each topic and selecting the one with the highest score as the bug type. Next, the approach assigns a score representing the bug-fixing cost for each developer based on their developer profile. Simultaneously, it employs content-based recommendation techniques on new and existing bug reports to assess developer experience. Each developer receives an experience score based on the word similarity between the new bug report and their previously fixed bug reports. Finally, these scores are combined and ranked in descending order to generate a list of suitable developers.

In cases where the bug type cannot be identified from the new bug report, CosTriage uses source code snippets from bug reports to make the determination. It selects 100 bug reports containing source code snippets and matches the import portions of source code between the new and existing reports using the Jaccard similarity coefficient. The bug type of the new report is then determined based on the type of the most similar existing bug report. However, CosTriage has limitations. It may fail when a new bug report lacks a source code

snippet. Additionally, the approach needs to guide handling new developers, as it relies on previous bug-fixing ratings that may not be available for newcomers. Therefore, it may not be easily generalized in an industrial context.

3.11 Industry-Oriented Approaches

Open-source and industrial projects have distinct characteristics, including development processes, requirements, and project sizes. While many bug-triaging approaches focus on open-source systems, recent studies have emphasized the growing need for effective bug assignment systems in industrial settings. Here, we explore key studies in this area.

3.11.1 Research-Industry Cooperation

To bridge the gap between academic research and industry requirements, Vaclav et al. [100] conducted a study that explored bug assignment automation in collaboration with both an industrial project (a Czech Republic-based software company) and an open-source project (Firefox). This study tested six hypotheses to compare bug assignment trends in these two domains, employing statistical methods like Chi-Square and t-tests. The findings indicated that the data distribution in the two datasets was similar, and a classification model using SVM + TF-IDF + stop word removal proved effective for both the industrial and Firefox data. This study shed light on the need to consider the number of issues per developer and the importance of supporting team recommendations in future research collaborations with companies.

3.11.2 Team Assignment

In the industry, there's a demand for recommendations for individual developers and developer teams to optimize resource utilization. Prior studies have highlighted the benefits of considering heterogeneous features in training models to improve recommendation accuracy. This approach involves constructing networks to represent relationships between various entities, such as bug reports and developers, fostering collaboration.

For team recommendations, Zhang et al. [101] introduced KSAP, a bug report assignment technique that utilizes K Nearest Neighbor (KNN) search and heterogeneous proximity. KSAP starts by building a heterogeneous network using existing bug reports involving five types of entities: developer, bug, comment, component, and product. These

entities are interconnected through various relations, such as developers writing comments or bugs containing comments. The study proposed nine meta-paths to identify developer collaborations, categorized into three types: associations on common bugs, components, and products. When a new bug report arrives, KSAP converts it into a document vector and calculates cosine similarities with existing bug reports. The K most similar bug reports are considered candidates, and the developers involved in activities related to these bugs are added to the list. The approach extracts each developer in the candidate list's associated meta-paths from the network. Finally, each developer's heterogeneous proximity score is computed based on these meta-paths, reflecting their collaboration with other developers on common bugs, components, and products. This score ranks the candidates, and the top Q developers are recommended.

However, KSAP faces challenges related to over-specialization, as it doesn't consider the latest developer activities. This leads to experienced developers being inundated with tasks while new developers need help to participate in bug resolution effectively.

3.12 Summary

The extensive exploration of existing bug-triggering techniques in the preceding discussion reveals a comprehensive body of research dedicated to automating this task. The literature describes ten distinct categories of work within automatic bug triggering, encompassing topic model-based, information retrieval-based, social network analysis-based, dependency-based, machine learning-based, reassignment-based, text categorization-based, data reduction-based, cost-aware-based, source-based, and industry-oriented-based approaches. However, it is notable that most of these approaches are tailored to address triaging requirements in open-source systems. One key observation is that existing solutions typically recommend developers based on either their prior bug history or recent source commits, often neglecting the complementary information present in the other source. This limitation results in imprecise recommendations and needs more collaboration among new developers.

Furthermore, a critical gap in the previous research is the need for load-balancing considerations, which are paramount in ensuring the equitable distribution of tasks and fostering collaboration among development teams. The oversight of these crucial factors renders the existing approaches less applicable to industrial projects. Consequently, there is a pressing need for further research to integrate and account for these vital aspects, thus establishing a more versatile and robust bug-triggering framework.

In light of these research gaps and the significance of load balancing in bug assignment, this study introduces two novel models. In Chapter 4, we present BSDRM to enhance developer recommendations by effectively leveraging both bug history and source commits. Subsequently, in Chapter 5, we introduce DevSched to address load-balancing concerns, ensuring a more equitable distribution of tasks among developers. Collectively, these innovations aim to resolve the identified issues and provide a comprehensive solution for bug assignment in industrial projects.

Chapter 4

Recommend Developer Team Efficiently

Bug triage is a critical process involving the prioritization of bugs based on various factors such as severity, frequency, and risk. This procedure is pivotal in justifying the allocation of resources for resolving different bug severities, ultimately leading to improved software quality within reduced timeframes. Many software companies grapple with the constant influx of a substantial volume of bug reports. When a new bug report surfaces, it typically necessitates the involvement of multiple adept developers for resolution. However, this often results in experienced developers being inundated with excessive bug assignments while newer and intermediate-skilled developers need help to secure opportunities for bug-fixing tasks. Additionally, situations may arise where experienced developers transition from different fields to undertake bug-fixing responsibilities. Therefore, a compelling need arises to allocate bugs to diverse categories of developers where they can contribute their knowledge and expertise effectively. To achieve this, we harness the available bug report information, encompassing aspects such as severity, priority, source code details, and commit logs, to identify suitable developers for addressing specific issues. Recent commits serve as valuable indicators of developers' ongoing activities, while historical records offer insights into a developer's proficiency in tackling particular bug types. However, it's essential to acknowledge that many freshly graduated developers, lacking prior bug-fixing experience, often join companies. In most cases, these newcomers may need to be better versed in such tasks.

Conversely, some developers have garnered valuable experience in resolving various bug types. Additionally, developers transitioning from other development projects are tasked with addressing incoming bugs, and both groups are considered new, experienced developers. Regrettably, they may not receive a sufficient volume of bugs to improve their skills.

Consequently, experienced developers become overwhelmed and need help to address numerous bugs within stipulated timeframes.

In response to these challenges, there is a need to create developer teams that encompass a mix of experienced professionals, fresh graduates, and developers with diverse backgrounds. This approach ensures that all categories of developers have the opportunity to contribute to the resolution of newly reported bugs. Moreover, collaborative problem-solving within such teams proves highly beneficial for addressing incoming bugs efficiently. Skilled developers can oversee a more extensive array of bugs, while newer additions to the team can learn and develop their bug-solving abilities. Despite prior research efforts focusing on bug-triaging methods, a significant gap still needs to be addressed to accommodate newly experienced developers adequately.

Machine learning, a field dedicated to developing algorithms that autonomously learn from past data to make informed decisions, is the foundation of our proposed solution. We introduce the Bug Solving Developer Recommendation Model, a Machine Learning-based approach designed to recommend developer teams consisting of experts, proficient developers from other domains, and fresh graduates to address newly reported bugs collaboratively.

4.1 Overview of BSDRM

In this section, we introduce an advanced Machine Learning-driven bug assignment method named BSDRM, designed to suggest a team of developers comprising individuals with varying expertise levels, including Expert Developers, Novice Expert Developers, and Fresh Graduates. Figure 4.1 provides an illustrative overview of the operational stages of this model, which will be briefly elaborated upon in the subsequent discussion.

4.1.1 Dataset Description

The datasets employed in various bug triage studies often use open-source projects such as Eclipse, Mozilla, and Netbeans. These datasets encompass an array of data types, including source code, commit logs, and bug reports sourced from Bugzilla. They encompass diverse variables, such as severity, priority, fix status, platform or hardware details, assignee information, details about a single developer, initial assignee, developer comments, summary (or title), and descriptions of bugs. The utilization of bug reports from the Eclipse, Mozilla, and NetBeans datasets in this study is substantiated by their widespread adoption as standard

benchmarks in the field. These datasets have been extensively employed in numerous prior studies [58-60, 84-88], establishing them as representative and well-established sources for bug-related research.

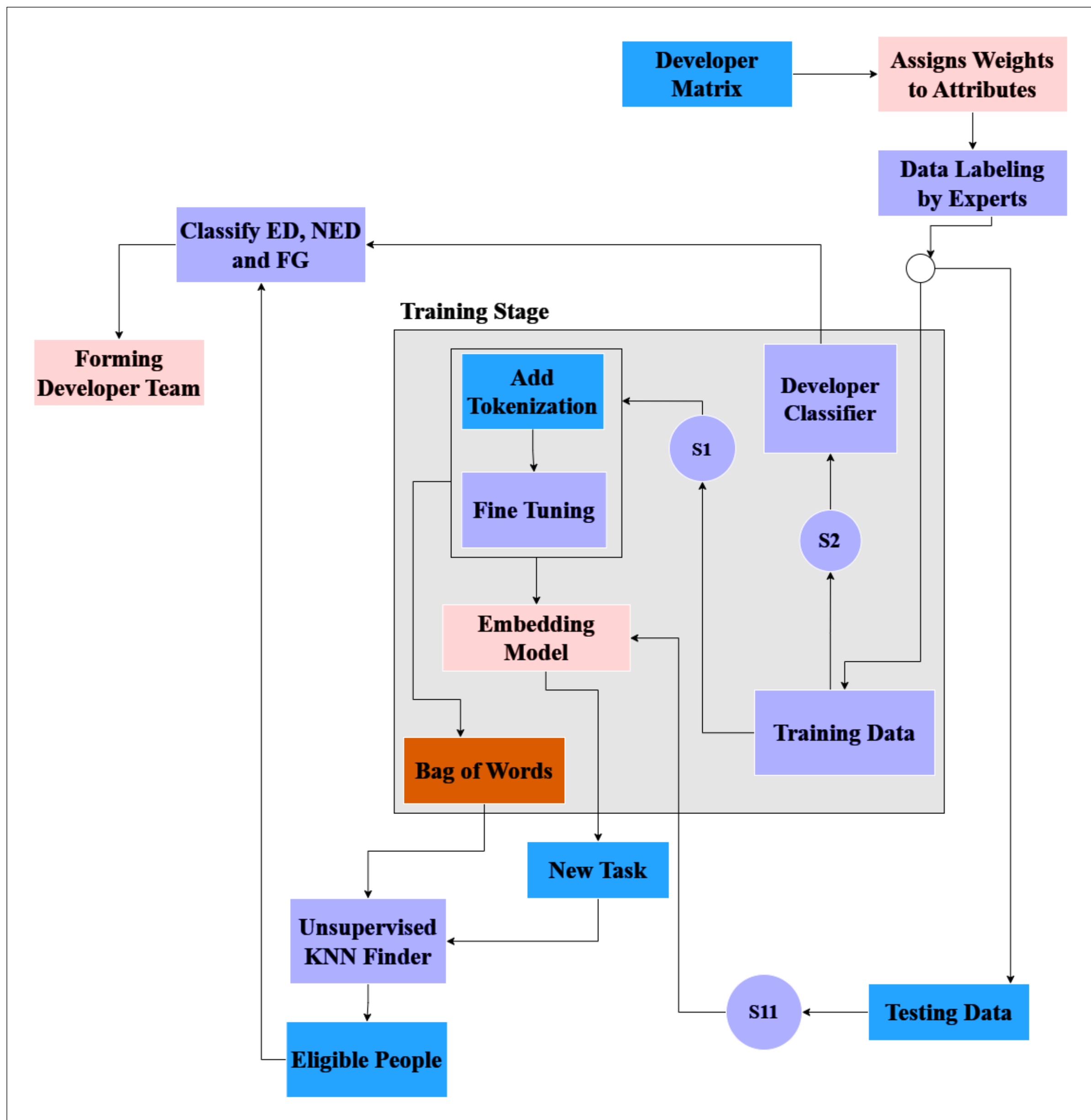


Figure 4.1: The workflow diagram of BSDRM, ML based developer recommendation model for bug triage

For our BSDRM investigation, we use this dataset to recommend suitable developers for bug resolution. Specifically, we have gathered a substantial volume of bug reports, amounting to 137,147 reports spanning from 2001 to 2020 for Eclipse, 132,261 reports ranging from 1999 to 2020 for Mozilla, and 44,149 reports covering the period between 2001 and 2017 for NetBeans. The summary of data for our BSDRM is presented in Table 4.1. The detailed descriptions of these datasets are described below:

4.1.1.1 Eclipse

Eclipse, one of the most renowned integrated development environments (IDEs), has emerged as a cornerstone in bug-triaging endeavors. Its significance in the realm of open-source projects lies in its extensive user base and its substantial contribution to bug reporting and resolution. Eclipse's bug reports, meticulously collected and analyzed as part of our BSDRM model, provide a comprehensive view of the challenges faced by developers and the diverse range of issues encountered in software development. With over 13, 71,147 bug reports gathered from Eclipse, our research delves deep into this treasure trove of data, spanning nearly two decades from 2001 to 2020. These bug reports encapsulate valuable insights into the software development process, including the severity and priority of issues, the status of fixes, the specific hardware or platform involved, the developers responsible, their comments, and concise summaries and descriptions of the reported problems. Leveraging this extensive dataset, our BSDRM model excels in intelligently recommending developer teams, comprising Experts in Development, Non-Experienced Developers, and Fresh Graduates, for efficient bug resolution within Eclipse and beyond.

Table 4.1: Dataset description

Datasets	Time Periods	Number of Reports
Eclipse	2001 to 2020	137,147
Mozilla	1999 to 2020	132,261
NetBeans	2001 and 2017	44,149

4.1.1.2 Mozilla

Mozilla, a prominent player in the world of open-source software, stands as another invaluable resource in the domain of bug-triaging research. Its bug reports have been instrumental in shedding light on the intricacies of software development challenges and bug resolution processes. Mozilla's bug repository, meticulously curated and scrutinized for our BSDRM model, provides a rich tapestry of insights into the multifaceted world of open-source development. With a staggering collection of over 13, 22,261 bug reports spanning the period from 1999 to 2020, Mozilla's dataset offers a substantial historical perspective on software issues. These bug reports encompass diverse attributes, including the criticality of issues, priority levels, fix statuses, hardware and platform details, assignees, initial developers assigned to the task, developer comments, concise yet informative summaries, and detailed descriptions of reported problems. This comprehensive dataset not only fuels the BSDRM

model's recommendations but also aids in understanding the complex dynamics of bug resolution within Mozilla's development ecosystem. Our model leverages this wealth of information to intelligently recommend developer teams, consisting of Experts in Development, Non-Experienced Developers, and Fresh Graduates, for efficient and effective bug resolution, thereby contributing significantly to the overall improvement of the Mozilla open-source project.

4.1.1.3 NetBeans

NetBeans, a renowned open-source integrated development environment (IDE), plays a pivotal role in advancing the field of bug-triaging research. Our exploration into NetBeans' bug repository has unearthed a treasure trove of valuable insights and data, which forms a cornerstone for the BSDRM model. With an impressive collection of 44,149 bug reports covering the period from 2001 to 2017, NetBeans' dataset enriches our understanding of software development challenges and bug resolution processes. These bug reports encompass many attributes, including severity levels, priority designations, fix status updates, hardware and platform specifications, assignees, the first developers assigned to the issues, developer comments, concise yet informative summaries, and comprehensive descriptions of the reported problems. NetBeans' dataset not only fuels the BSDRM model's recommendations but also serves as a lens through which we gain deeper insights into the nuanced world of open-source software development. By harnessing this extensive dataset, our model provides intelligent recommendations for developer teams consisting of Experts in Development, Non-Experienced Developers, and Fresh Graduates. These recommendations significantly enhance the bug resolution processes within the NetBeans open-source project, ultimately fostering its growth and success in the software development landscape.

4.1.2 Generating Developer Matrix

Creating the Developer Matrix involves merging datasets from Eclipse, Mozilla, and NetBeans, resulting in a comprehensive dataset comprising 56,621 instances, each representing a developer. We introduce three distinct profiles to evaluate and categorize their skills: Experts in Development, Non-Experienced Developers, and Fresh Graduates. In most of the works, experienced developers who solved many problems were assigned to fix different types of bugs. Thus, we scrutinized the literature [66, 68-70] and found some limitations in the selection of developers. In previous works, they mainly mentioned

experienced developers and fresher. However, many developers can be categorized as mid-category in the real industrial sector. In addition, some experts come from other technical domains and have expertise in solving any kind of problem. Thus, these kinds of developers can be employed to fix different bugs and mitigate the limitations of existing works.

4.1.2.1 ED Profile

Experts in Development are seasoned professionals in the realm of bug resolution. They are known for their extensive experience and exceptional problem-solving skills. EDs are characterized by their ability to single-handedly tackle a substantial volume of bugs single-handedly, demonstrating a high level of independence in their work. They actively discuss issues, offering valuable insights and solutions to complex problems. EDs are frequently designated as the initial developers for many bugs, showcasing their reliability and expertise in handling critical issues. These developers are the backbone of bug triage teams, often setting the standard for bug resolution efficiency. Specifically, an ED is identified as someone who has fixed a minimum of 200 bugs, engaged in discussions on 1000 or more issues, and served as the first developer for around 500 bugs.

4.1.2.2 NED Profile

Non-experienced developers represent a diverse group with varying levels of expertise. These developers typically fall into two categories. First, some are in the early stages of their bug-fixing careers and, while still being experts, have moderate experience in resolving bugs. They may have transitioned from other software domains, bringing valuable skills and perspectives to bug triage. Second, NEDs include developers who, while experienced in other areas, are temporarily working on bug-fixing tasks. This category acknowledges their adaptability and the potential to contribute effectively to bug resolution efforts. NEDs typically handle approximately 100-200 bug resolutions, comment on fewer than 1000 bug reports, and are assigned as the first developer for fewer than 500 bugs.

4.1.2.3 FG Profile

Fresh Graduates are developers who have recently completed their academic studies and embarked on their careers in bug resolution. They represent a unique category in the bug triage ecosystem. Given their limited professional experience, their work history within existing bug repositories may be minimal or nonexistent. Therefore, assessing the skills and

capabilities of FGs involves factors beyond traditional bug-fixing metrics. Authorities may provide a predefined evaluation form to holistically evaluate FG skills, encompassing criteria such as academic knowledge, technical expertise, interests, and any projects they may have undertaken during their educational journey.

Assigning weights to individual features based on their significance is a critical step in determining each developer's expertise level, emphasizing qualitative evaluation over purely quantitative methods. To ensure the accuracy of this labeled dataset, it undergoes a review process by domain specialists. Subsequently, the dataset is divided into training and testing sets, with the training set further segmented into two subsets: Subset-1 (S1) containing summary and description attributes and Subset-2 (S2) comprising the remaining attributes such as severity, priority, fix status, platform or hardware, assignee, single developer, first assignee, commenter, and developers. A Subset-11 (S11) is also extracted from the test set, focusing on summary and description attributes for comprehensive analysis and evaluation. This meticulous process ensures the robustness and reliability of the BSDRM model's developer categorization and recommendation capabilities.

4.1.3 Training Stage

During the training stage of the BSDRM, we consolidate Subset-1 and Subset-2 to facilitate the training of sentence-embedded models employing diverse classifiers. This pivotal stage involves various training procedures instrumental in forming an adept developer team. The ensuing steps in this session are succinctly elucidated as follows:

4.1.3.1 Sentence Embedding

Initially, we initiate a fine-tuning procedure on Subset-1 using a pre-trained Bidirectional Encoder Representations from Transformers (BERT) model to construct a sentence embedding model [102, 103]. BERT is a state-of-the-art natural language processing model developed by Google [104, 105]. It has revolutionized various NLP tasks by pre-training on a massive corpus of text and then fine-tuning it for specific tasks. BERT belongs to a family of models known as Transformers, which have demonstrated exceptional performance in understanding the context and semantics of natural language. BERT is bidirectional, which allows it to consider both the left and right context of a word in a sentence. This bidirectional understanding is crucial for comprehending the meaning of a word in the context of a sentence, as many words rely on their surrounding words for interpretation.

The pre-training phase involves exposing BERT to a vast amount of text data, such as books, articles, and websites, to learn the relationships between words, phrases, and sentences. This process enables BERT to capture semantic nuances and contextual information, making it a powerful tool for various NLP tasks, including text classification, sentiment analysis, question answering, and, in this case, sentence embedding. In the context of sentence embedding, BERT takes a sentence or a piece of text and transforms it into a fixed-length vector representation. This representation encodes the meaning and context of the text in a dense numerical format. These sentence embeddings can then be used for various downstream tasks, such as developer team recommendation, by measuring the similarity between sentences or texts.

Fine-tuning BERT on specific datasets or tasks allows it to adapt to the requirements of those tasks. This fine-tuning process involves training the model on labeled data related to the target task. For example, in the case of developer team recommendation, BERT can be fine-tuned using data specific to bug triage and developer expertise. This model constructs a collection of words associated with the developer within this framework by conserving the contextual relevance of words from S1. In instances where unknown words are detected, they are seamlessly integrated into the BERT.

4.1.3.2 Balancing Data

Within S2, the uneven distribution of different developer types has introduced a significant imbalance in our dataset, which brings about a range of inherent disadvantages. This data disproportionality poses notable challenges, particularly in machine learning applications. Overfitting, a common issue when dealing with imbalanced data, becomes a concern as models may need help to generalize effectively to underrepresented developer categories, resulting in suboptimal predictive performance. Furthermore, imbalanced data can lead to biased model outcomes, where the majority class tends to dominate predictions at the expense of minority classes.

To address these challenges, we have proactively taken steps to resample and equalize this specific subset within our dataset. We create a balanced representation of the data using resampling by ensuring that precisely 652 samples represent each developer type. This rebalancing effort yields several key advantages. Firstly, it fosters improved model performance, as the models can now learn from a more representative distribution of developer types. This leads to more accurate and reliable recommendations for developers

across the expertise spectrum.

Moreover, balancing the dataset mitigates the potential for bias and discrimination in our models, ensuring that developers of all categories receive equitable opportunities and recommendations. Additionally, it enhances the robustness of our models, allowing them to handle real-world scenarios with varying proportions of developer types. This means that our models can maintain their effectiveness even when faced with new and unseen data. Addressing the data imbalance within S2 is a crucial step in ensuring the reliability and fairness of our machine-learning models. By achieving a balanced representation, we overcome challenges associated with imbalanced data, enhance model performance, reduce bias, and promote equitable recommendations for developers with diverse expertise levels. This strategic data handling approach strengthens the overall utility of our models in the context of bug assignment and developer recommendation.

4.1.3.3 Employing Different classifiers

We proceed to train various classifiers on S2, including Decision Tree, Extra Tree, AdaBoost, Bagging Classifier, Gradient Boosting, KNN, Nearest Centroid, Bernoulli Naïve Bayes, Multinomial Naïve Bayes, Complement Naïve Bayes, Gaussian Naïve Bayes, Logistic Regression, Perceptron, and Multi-Layer Perceptron. These classifiers collectively form a developer classifier designed to categorize new records based on their experience level. This section presents a detailed description of these classifiers.

a) Decision Tree

A Decision Tree is a popular and intuitive ML algorithm for classification and regression tasks. It models decisions or predictions by recursively breaking down a complex problem into a series of simpler decisions based on input features. The tree-like structure resembles a flowchart where each internal node represents a decision or test on a feature, each branch represents an outcome of that test, and each leaf node represents the final prediction or class label. Constructing a decision tree involves selecting the most informative features and splitting the data into subsets to make decisions. The process aims to maximize the data's homogeneity (or purity) within each subset while minimizing impurity. Common impurity measures for classification tasks include Gini impurity and entropy, while Mean Squared Error is used for regression tasks.

Decision Trees are easy to interpret and visualize, making them valuable for understanding the logic behind a model's predictions. However, they are prone to overfitting when the tree becomes too deep and complex. To mitigate this, techniques like pruning can

be applied to simplify the tree by removing nodes that do not contribute significantly to predictive accuracy. Decision Trees find applications in various fields, including finance for credit scoring, healthcare for disease prediction, and natural language processing for sentiment analysis [106-108]. Their simplicity, interpretability, and effectiveness make them a valuable tool in the machine learning toolkit.

b) Extra Tree

Extra Trees, short for Extremely Randomized Trees, is a powerful ensemble learning technique rooted in decision tree algorithms. What sets Extra Trees apart from traditional decision trees is its remarkable level of randomness during the tree-building process. Unlike regular Decision Trees, which carefully evaluate features and thresholds to split nodes, Extra Trees introduces even more randomness by selecting random subsets of features and thresholds at each node. This randomness makes Extra Trees significantly less prone to overfitting, enhancing its ability to generalize well to unseen data.

One of the key advantages of Extra Trees is its robustness in noisy datasets and high-dimensional spaces. Incorporating additional randomness makes it more resistant to outliers and noisy data points, resulting in more reliable and accurate predictions. Extra Trees also harnesses the power of ensemble learning through techniques like bagging, where multiple decision trees are constructed on different subsets of the training data. The outputs of these trees are combined to provide the final prediction, which is often achieved by averaging the results for regression tasks and majority voting for classification tasks. Overall, Extra Trees is a valuable tool in machine learning that strikes a balance between reducing overfitting and improving model performance, making it a popular choice in various data-driven applications [109].

c) AdaBoost

AdaBoost, short for Adaptive Boosting, is a popular ensemble learning algorithm that focuses on improving the performance of weak learners to create a strong, accurate predictive model. AdaBoost combines multiple weak learners, often simple decision trees or stump classifiers, into a weighted ensemble. The key idea behind AdaBoost is to give more weight to those training samples that the current weak learners misclassify during each iteration. By emphasizing the mistakes, AdaBoost encourages the subsequent weak learners to focus on the previously misclassified data points, progressively improving the model's accuracy.

AdaBoost works iteratively, with each iteration introducing a new weak learner. The algorithm assigns higher weights to the samples misclassified by the previous weak learners and lower weights to those classified correctly. As a result, the subsequent weak learners

concentrate more on the challenging samples. AdaBoost calculates the performance of the newly added weak learner and adjusts the weights accordingly. This process continues for a predefined number of iterations or until the model reaches a desired level of accuracy. One of the significant advantages of AdaBoost is its ability to adapt and excel in a wide range of machine-learning tasks, including classification and regression. It is beneficial when dealing with complex datasets, as it can reduce overfitting and provide robust generalization. However, AdaBoost can be sensitive to noisy data and outliers, and its performance may degrade in the presence of such anomalies. Overall, AdaBoost is a versatile ensemble learning technique that has proven effective in various real-world applications thanks to its ability to enhance the performance of weak learners and create strong predictive models [110-112].

d) Bagging Classifier

The Bagging Classifier, short for Bootstrap Aggregating Classifier, is a powerful ensemble learning technique that enhances the accuracy and robustness of machine learning models. Bagging creates multiple subsets of the training data through bootstrap sampling, where data points are randomly selected with replacement. Each subset trains an independent base learner, typically a decision tree.

By leveraging the collective knowledge of multiple base models, Bagging reduces the risk of overfitting and increases the model's overall stability and predictive performance. It is particularly effective when dealing with noisy or complex datasets, as it minimizes the influence of individual outlier data points. Popular variations of Bagging include the Random Forest algorithm, which introduces an additional layer of randomness by considering only a random subset of features during each tree's construction, further enhancing model diversity and robustness. In essence, Bagging Classifier harnesses the power of ensemble learning to improve the accuracy and resilience of machine learning models by combining multiple base learners into a unified, more reliable predictor. Its versatility and effectiveness make it a valuable tool in various domains, including classification, regression, and feature selection [112-114].

d) Gradient Boosting

Gradient Boosting is a powerful ensemble learning technique used in machine learning for building predictive models. It is particularly effective in solving regression and classification problems. Unlike Bagging, which creates multiple independent base models in parallel, Gradient Boosting produces a sequence of models iteratively, with each new model aiming to correct the errors made by the previous ones. It starts with an initial model, often a simple

one like a decision tree, which serves as the first base learner. A new base learner is added to the ensemble in each subsequent iteration. This new learner is trained on the errors or residuals of the previous models. The idea is to "boost" the performance by focusing on previous models' challenging examples. The predictions of each base learner are weighted and combined. These weights are determined during the training process, where the goal is to minimize the overall error. Gradient Boosting uses a gradient descent optimization technique to find the best weights and model parameters. It calculates the gradient of the loss function concerning the ensemble's output and adjusts the weights and parameters accordingly. The process continues for a specified number of iterations or until a stopping criterion is met, such as reaching a minimum error or a maximum number of models.

Gradient Boosting has several popular implementations, including Gradient Boosting Machines (GBM), XGBoost, LightGBM, and CatBoost. These implementations optimize the algorithm for efficiency and often include additional features like regularization to prevent overfitting. The strength of Gradient Boosting lies in its ability to create highly accurate predictive models by focusing on complex examples. However, it can be computationally intensive and may require careful tuning of hyperparameters to achieve the best results. It's widely used in various applications, including ranking, recommendation systems, and many Kaggle competitions [116, 117].

e) K-Nearest Neighbors

A flexible machine-learning technique for classification and regression applications is K-Nearest Neighbors. Its fundamental principle relies on proximity-based prediction: it predicts the class or value of a data point based on the majority class or average value of its nearest neighbors within the training dataset. K-NN is a straightforward algorithm, making it an attractive choice for various applications. In K-NN, the process begins with a dataset containing labeled data points, where each data point is represented as a vector of features and is associated with a class label (for classification) or a target value (for regression). One essential hyperparameter to set is "K," which represents the number of neighbors to consider when making predictions. The choice of K can significantly impact the algorithm's performance.

The algorithm employs a distance metric, such as Euclidean or Manhattan distance, to measure the similarity or dissimilarity between data points. The K-NN model identifies the K-nearest neighbors in the training dataset based on the selected distance metric. For classification tasks, it tallies the occurrences of each class among these neighbors and assigns the class with the highest count as the predicted class for the new data point. In regression

tasks, K-NN computes the average (or weighted average) of the target values of the K-nearest neighbors and assigns this average as the predicted value for the new data point.

K-NN's simplicity and intuitive approach make it a popular choice for many applications, including recommendation systems, image classification, and anomaly detection [118-120]. However, it has some limitations, such as sensitivity to the choice of K and the need for an extensive training dataset to provide reliable predictions. Additionally, it can be computationally expensive for large datasets because it requires calculating distances between the new data point and all points in the training set.

f) Nearest Centroid

Often employed for classification tasks, the Rocchio method, known as the Nearest Centroid algorithm, is an easy-to-understand machine learning technique. It is useful when dealing with text classification problems but can also be applied to other domains. In the Nearest Centroid algorithm, each class or category is represented by a centroid, essentially the mean vector of all the data points belonging to that class in the feature space. To build these centroids, you start with a labeled dataset where each data point is associated with a class label. For each class, you calculate the mean of the feature vectors of all data points, resulting in a centroid vector representing the class. To make a prediction for a new, unlabeled data point, the algorithm calculates the Euclidean distance (or another chosen distance metric) between the new data point and each class's centroid. The class, whose centroid is closest to the new data point, as determined by the distance calculation, is assigned as the predicted class for that data point.

The Nearest Centroid algorithm is relatively simple and computationally efficient, making it a good choice for text classification tasks, especially when dealing with high-dimensional data such as text documents. However, it may perform less well as more complex algorithms in scenarios where the decision boundaries between classes are nonlinear or intricate. Despite its simplicity, the Nearest Centroid algorithm can be surprisingly effective in various applications, including spam detection, document categorization, and sentiment analysis [121-122].

g) Bernoulli Naïve Bayes

Bernoulli Naïve Bayes is a probabilistic machine learning algorithm and a variant of the Naïve Bayes classifier. It is particularly suited for binary classification tasks, where the goal is to categorize data into one of two classes, typically denoted as "positive" and "negative" or "1" and "0." This algorithm is commonly used in text classification, sentiment analysis, and spam detection, where the input features often represent the presence or absence of specific binary attributes. The Bernoulli Naïve Bayes classifier is based on the principles of Bayes'

theorem and the assumption of feature independence within each class. It models the conditional probability of a document belonging to a specific class (e.g., spam or not spam) given the presence or absence of particular binary features (e.g., the occurrence of certain words). To make predictions, Bernoulli Naïve Bayes calculates the likelihood of observing each feature in both the positive and negative classes and combines this information with prior probabilities to estimate the final class probabilities. It then assigns the class with the highest probability as the predicted class for the input data point.

One key characteristic of Bernoulli Naïve Bayes is that it treats input features as binary variables, representing whether a particular attribute is present or not. This makes it suitable for tasks where only the presence of features matters and ignores their frequencies or numerical values. While it's less expressive than other Naïve Bayes variants like Multinomial Naïve Bayes, it can be highly effective in scenarios with sparse binary data, such as text documents. Bernoulli Naïve Bayes provides a simple yet robust way to perform binary classification tasks, especially when dealing with text data and situations where feature independence assumptions are reasonable [123, 124].

h) Multinomial Naïve Bayes

Multinomial Naïve Bayes is a popular machine learning algorithm used primarily for text classification and natural language processing tasks. It's an extension of the Naïve Bayes algorithm. It is particularly well-suited for situations where features represent discrete counts or frequencies, such as word occurrences in documents or term frequencies in text data. The Multinomial Naïve Bayes classifier is based on the principles of Bayes' theorem and the assumption of feature independence within each class. It's commonly applied in tasks like spam detection, sentiment analysis, document categorization, and topic classification.

In text classification, documents are typically represented as vectors of term frequencies or other similar representations where each feature corresponds to a unique word or term in the entire corpus. The Multinomial Naïve Bayes model calculates the conditional probability of a document belonging to a particular class given the frequencies of terms in that document. To make predictions, Multinomial Naïve Bayes estimates the likelihood of observing each term in both the positive and negative classes and combines this information with prior probabilities to calculate the final class probabilities. It then assigns the class with the highest probability as the predicted class for the input document. One key advantage of Multinomial Naïve Bayes is its ability to handle features representing counts or frequencies. This makes it suitable for text data where words are counted, and their occurrences matter. It's effective in scenarios where feature independence assumptions are reasonable and

performs well even with relatively small training datasets. Multinomial Naïve Bayes is a robust and widely used algorithm in text classification tasks, mainly when dealing with high-dimensional feature spaces and discrete count-based representations of data [125-127].

i) Complement Naïve Bayes

Complement Naïve Bayes is a variation of the traditional Naïve Bayes classifier designed to address imbalanced datasets, where one class significantly outnumbers the other(s). While the standard Multinomial Naïve Bayes tends to favor the majority class, CNB is tailored to give more accurate predictions in situations where class distribution is skewed. The key idea behind Complement Naïve Bayes is to complement the standard MNB model by considering the distribution of features in the minority class relative to the majority class. CNB starts by calculating the probabilities of features within each class, just like MNB. However, instead of focusing on the class of interest (the positive class), it computes probabilities for the complementary class (the negative class). This means it estimates how likely features appear in documents not belonging to the class of interest. CNB then selects the class with the lowest complementary probability as the predicted class for a given document. In other words, it looks for the class least likely to contain the observed features, making it useful for imbalanced datasets where the minority class is of interest.

CNB is particularly effective for text classification tasks, such as spam email detection, where the spam class (the minority) is the focus [128]. It can help mitigate the class imbalance problem by giving more importance to the underrepresented class during classification. CNB has been shown to perform well in scenarios where traditional MNB or other classifiers may struggle due to skewed class distributions. Complement Naïve Bayes is an extension of the Naïve Bayes algorithm that excels in handling imbalanced datasets by considering feature probabilities in the complementary class and is commonly used in text classification tasks where class imbalance is a challenge.

j) Gaussian Naïve Bayes

Gaussian Naïve Bayes (GNB) is a variant of the Naïve Bayes algorithm that is specifically designed for continuous data or features that can be modeled using a Gaussian (normal) distribution. Unlike the standard Naïve Bayes, suitable for discrete data like text, GNB assumes that the features follow a Gaussian distribution within each class. GNB begins by estimating the probability density function (PDF) for each feature within each class. It assumes that the data for each feature in a given class follows a Gaussian distribution. This means that it calculates the mean (average) and variance (spread) of the values of each feature for each class. Like the standard Naïve Bayes, GNB also assumes that features are conditionally independent within each class. This simplifying assumption allows GNB to

calculate the joint probability of observing a set of feature values for a given class. GNB calculates each class's prior probability, representing the likelihood of encountering each class in the dataset. To classify a new data point, GNB applies Bayes' theorem to calculate the posterior probability of each class given the observed feature values. It then assigns the class with the highest posterior probability as the predicted class for the data point.

Gaussian Naïve Bayes is instrumental when dealing with continuous or real-valued data, such as measurements, sensor data, or scientific data [129-131]. It assumes that the features within each class are typically distributed and uses this assumption to estimate probabilities and make predictions. One of the advantages of GNB is its simplicity and efficiency in high-dimensional feature spaces. However, it may not perform well if the Gaussian assumption does not hold for the data or if features are strongly correlated. Other classifiers like Support Vector Machines or Decision Trees may be more appropriate in such cases.

k) Logistic Regression

Logistic Regression is a popular and widely used statistical method for binary classification, which means it's primarily used when you want to predict one of two possible outcomes (e.g., yes/no, spam/not spam, pass/fail). Despite its name, logistic Regression is a classification algorithm rather than a regression algorithm used for continuous prediction. Using the logistic function, it models the relationship between the binary dependent variable (the target class) and independent variables (features). The logistic function maps any real-valued number to a value between 0 and 1, which can be interpreted as a probability. The logistic function equation is presented by Equation (4.1):

$$P(Y=1|X) = \frac{1}{1+e^{-z}} \qquad (4.1)$$

Where:

P(Y=1|X) is the probability that the dependent variable Y is equal to 1 (the positive class) given the values of the independent variable X.

e is the base of the natural logarithm (approximately equal to 2.71828).

z is the linear combination of the feature values and model coefficients: $z = b_0 + b_1X_1 + b_2X_2 + ... + b_nX_n$.

Logistic Regression estimates the coefficients (b0, b1, b2, ..., bn) best fitting the data in the training phase. This process involves using a method like Maximum Likelihood Estimation (MLE) to find the values of the coefficients that maximize the likelihood of the observed outcomes given the model. Once the model is trained, you can use it to make predictions. Logistic Regression calculates the probability that a new data point belongs to the positive class (P(Y=1|X)). If this probability is

more significant than a certain threshold (typically 0.5), the model predicts the positive class; otherwise, it predicts the negative class. Key characteristics and advantages of Logistic Regression include its simplicity, interpretability, and the ability to model linear and nonlinear relationships between features and the log-odds of the target class. It's a valuable tool for problems where you must make binary decisions based on input features, such as credit scoring, medical diagnosis, and spam detection [132, 133]. However, Logistic Regression has limitations, such as its inability to handle more than two classes without modification (multinomial logistic Regression is used for that), and it may not perform well when the relationship between features and the log-odds of the target class is highly nonlinear. More complex models like decision trees or neural networks may be more appropriate in such cases.

l) Perceptron

A Perceptron is one of the simplest forms of artificial neural networks, serving as the foundational building block for more complex neural network architectures. Developed by Frank Rosenblatt in the late 1950s, the perceptron is designed for binary classification tasks, where it can determine whether an input belongs to one of two classes (e.g., yes/no, true/false) [134]. The perceptron takes multiple binary or real-valued inputs ($X_1, X_2, ..., X_n$). Each input is associated with a weight ($W_1, W_2, ..., W_n$), representing the connection's importance or strength between the input and the perceptron. The inputs are multiplied by their corresponding weights, and the weighted sum of these products is calculated by using Equation (4.2):

$$\text{Weighted Sum} = (X_1 * W_1) + (X_2 * W_2) + ... + (X_n * W_n) \tag{4.2}$$

The weighted sum is then passed through an activation function (typically a step or threshold function). The activation function determines the output of the perceptron. In the case of a step function, if the weighted sum is greater than or equal to a certain threshold (typically 0), the perceptron outputs 1 (representing one class); otherwise, it outputs 0 (representing the other class). Alternatively, a threshold function may be used, where the perceptron outputs 1 if the weighted sum is greater than or equal to the threshold and 0 otherwise. In addition to the inputs and weights, a bias term (often denoted as b or W0) is included. The bias allows the perceptron to adjust the output even when all inputs are zero. Mathematically, it shifts the decision boundary and can be expressed using Equation (4.3).

$$\text{Weighted Sum} = (X_1 * W_1) + (X_2 * W_2) + ... + (X_n * W_n) + b \tag{4.3}$$

The Perceptron's learning process involves adjusting the weights and bias to minimize classification errors. A training algorithm, like the perceptron learning rule, updates the weights and bias based on the misclassification of data points. The goal is to find the optimal weights and bias that allow the Perceptron to classify the training data correctly. While Perceptrons are powerful for simple

linearly separable problems, they have limitations. They can only solve linearly separable tasks, which means they cannot handle problems where classes are not separated by a straight line or plane. More complex neural network architectures, such as multilayer Perceptrons (Feed-Forward Neural Networks), were developed to perform nonlinear classification tasks to address this limitation. These networks consist of multiple layers of interconnected Perceptrons and can learn complex patterns from data.

m) Multi-Layer Perceptron

A Multi-Layer Perceptron is an artificial neural network for solving complex machine-learning problems, including classification and regression tasks. It is characterized by its multilayered architecture, consisting of an input layer, one or more hidden layers, and an output layer. MLPs are part of the broader family of Feed-Forward Neural Networks, where information flows in one direction, from input to output.

The input layer consists of neurons (also called nodes) corresponding to the features of the input data. Each neuron represents an input feature, and the input values are fed directly into these neurons. There is no computation within the input layer; it only passes the data to the subsequent layers. There may be one or more hidden layers between the input and output layers of an MLP. Neurons in these hidden layers process the input data through calculations. After receiving inputs from the preceding layer, each neuron in a hidden layer computes the weighted sum of these inputs and then an activation function. The number of neurons in each hidden layer and the number of hidden layers are hyperparameters that can be adjusted based on the complexity of the problem. Each connection between neurons (synapse) has an associated weight, which determines the strength of the connection. Additionally, each neuron in the hidden and output layers has a bias term, allowing the network to adjust even when the inputs are zero. Weights and biases are learned during the training process to optimize the network's performance. Activation functions introduce nonlinearity into the network, enabling MLPs to model complex relationships in the data. Common activation functions include the sigmoid (logistic) function, hyperbolic tangent (tanh) function, and rectified linear unit (ReLU) function. These functions introduce nonlinear transformations to the weighted sum of inputs, allowing the network to learn nonlinear patterns. The output layer produces the network's predictions or classifications based on the computations performed in the hidden layers. The number of neurons in the output layer depends on the nature of the task. For binary classification, a single neuron with a sigmoid activation function is often used, while for multiclass classification, there is typically one neuron per class with softmax activation. During inference or prediction,

forward propagation is the process of passing input data through the network to compute an output. This involves calculating the weighted sum of inputs, applying the activation functions in each layer, and producing the final output. Training an MLP involves adjusting the network's weights and biases to minimize a loss function that quantifies the error between the predicted outputs and the true labels. Backpropagation is the process of propagating this error backward through the network, layer by layer and updating the weights and biases using optimization algorithms like gradient descent. Techniques like dropout, L1 and L2 regularization, and various optimization algorithms (e.g., Adam, RMSprop, stochastic gradient descent) are commonly used to improve the generalization and training efficiency of MLPs.

MLPs are highly flexible and can approximate complex functions, making them suitable for various applications, including image and speech recognition, natural language processing, and predictive modeling [135-136]. However, their performance often depends on factors such as architecture design, hyperparameter tuning, and the quantity and quality of training data.

4.1.4 New Task

In this section, we focus on a critical task: assigning the most suitable developers to newly submitted bug reports and building efficient developer teams to resolve these issues. To commence this process, we direct our attention to S11, a selected subset that constitutes 20% of the testing reports designated for this specific purpose. Leveraging the capabilities of S11, we use a pre-trained sentence embedding model, a potent tool in natural language processing. Our primary aim is twofold: firstly, to gain a clear and contextually aware comprehension of the bug reports contained in S11, and secondly, to create a specialized vocabulary list tailored for developers. This developer-oriented vocabulary list is a crucial asset in our mission to adeptly match developers with bugs, ensuring that the language used aligns seamlessly with the nuances of bug resolution. It plays a pivotal role in forming teams optimized for efficient bug resolution.

4.1.5 Exploring Eligible Developer

To identify the most suitable developers for each bug report, we turn to the power of unsupervised learning by employing a K-Nearest Neighbors finder. This sophisticated tool acts as our guide in the developer assignment process. It compares the specialized vocabulary

list associated with the testing reports to the comprehensive bag of words compiled from developers' past interactions and contributions. By leveraging this approach, we can pinpoint the K nearest developers who exhibit the most pertinent expertise and experience to tackle the intricacies of the bug in question. This precise matching mechanism ensures that the right developers are selected for each bug report, thereby enhancing the overall efficiency and effectiveness of the bug resolution process.

4.1.6 Classifying Developers

Once we've identified the pool of eligible developers, our next crucial step involves categorizing them according to their experience levels. To accomplish this, we turn to the trusty developer classifier model, a fundamental component of our bug assignment system. The accuracy and reliability of this model depend on the collective performance of the individual classifiers operating within it. To evaluate the effectiveness of our developer classification, we subject it to rigorous scrutiny using a range of evaluation metrics. These metrics serve as our yardstick for measuring the model's proficiency in accurately categorizing developers based on their experience, ensuring that our bug resolution teams are well-balanced and composed of developers with the right skill sets.

4.1.6.1 Evaluation Metrics

We employ a valuable tool known as the confusion matrix to evaluate the effectiveness of our bug assignment system and developer classification model. This matrix offers a comprehensive view of the results, distinguishing between correct assignments (true positives - TP), misclassifications, which can be false positives (FP) or false negatives (FN), and cases correctly unassigned (true negatives - TN). To thoroughly assess our model's performance, we utilize diverse evaluation metrics, each providing unique insights into its effectiveness. These metrics encompass Accuracy (Accu), Precision (Prec), Recall (Rec), F1-Score (FS1), Hamming Loss (HL), Jaccard Score (JS), Matthews Correlation Coefficient (MCC), Area Under the Curve (AUC), and Cohen's Kappa Score (CKS). This array of metrics ensures a comprehensive evaluation of the classification results, guaranteeing that our system not only assigns developers accurately but also maintains the desired balance within bug resolution teams.

i) Accuracy: Accuracy is a straightforward metric that calculates the ratio of correctly predicted instances to the total number of instances in the dataset. It provides a general

measure of how well the classifier performs in terms of overall correctness. It is suitable for balanced datasets but can be misleading in imbalanced scenarios. It can be calculated using Equation (4.4).

$$\text{Accu} = \frac{TP+TN}{TP+TN+FP+FN} \tag{4.4}$$

ii) Precision: The classifier's positive predictions are the main focus of precision. It is the proportion of correctly predicted positive outcomes to all positive predictions (false positives plus true positives). Because precision measures the accuracy of positive predictions, it is especially crucial when the cost of false positives is significant. It can be expressed using Equation (4.5).

$$\text{Prec} = \frac{TP}{TP+FP} \tag{4.5}$$

iii) Recall: Recall evaluates the classifier's capacity to recognize every instance of a specific class; it is sometimes referred to as sensitivity or true positive rate. It is the ratio (true positives plus false negatives) of true positive predictions to that class's total number of real instances. Remembering is essential since there are severe repercussions for missing any favorable event. Equation (4.6) is the recall calculation formula.

$$\text{Rec} = \frac{TP}{TP+FN} \tag{4.6}$$

iv) F1-Score: The F1-Score is a balance between precision and recall, as shown in Equation (4.7). It calculates the harmonic mean of these two metrics, providing a single value that considers both false positives and false negatives. It is beneficial when there is an imbalance between classes.

$$\text{FS1} = \frac{2TP}{2TP+FP+FN} \tag{4.7}$$

v) Hamming Loss: Hamming Loss is specific to multi-label classification problems. It measures the fraction of incorrectly predicted labels across all instances, as illustrated in Equation (4.8). It quantifies the extent to which the classifier deviates from the correct labeling of instances.

$$\text{HL} = \frac{\sum_{j=1}^{N} XOR\,(I^j - P^j)}{N} \tag{4.8}$$

Where:

N is the total number of samples or instances.

$XOR\ (I^j - P^j)$ computes the element-wise exclusive OR (XOR) operation between the true labels (I^j) and the predicted labels (P^j).

vi) Jaccard Score: The Jaccard Score, also known as the Jaccard Index, assesses the similarity between two sets, as shown in Equation (4.9). In the context of multi-label

classification, it evaluates the agreement between the predicted set of labels and the actual set of labels for each instance. It quantifies how well the classifier captures the true labels.

$$JS = \frac{A \cap B}{A \cup B} \quad (4.9)$$

vii) Matthews Correlation Coefficient: MCC is a correlation coefficient that considers all four categories in the confusion matrix: true positives, true negatives, false positives, and false negatives. It ranges from -1 to 1, where 1 indicates perfect agreement between predictions and actuals, 0 suggests no agreement beyond chance, and -1 indicates complete disagreement. It can be expressed using Equation (4.10).

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)\ (TP+FN)+(TN+FP)\ (TN+FN)}} \quad (4.10)$$

viii) Area Under the Curve: The AUC metric is frequently employed when performing Receiver Operating Characteristic (ROC) analysis for binary classification. It measures how well the classifier can discriminate between the positive and negative classifications. Higher AUC values indicate better discrimination; values range from 0 to 1. It can be represented with TP rate (TPR) and T N rate (TNR) by the Equation (4.11):

$$AUC = \frac{TPR+TNR}{2} \quad (4.11)$$

ix) Balanced Accuracy: Balanced accuracy is particularly useful in multi-class classification scenarios with imbalanced class distributions. It calculates the average accuracy across all classes, ensuring that each class contributes equally to the overall accuracy score. This metric provides a more reliable assessment when classes are imbalanced. It can be represented by Equation (4.12).

$$BA = \frac{1}{2} \times \frac{TP}{TP+FN} + \frac{1}{2} \times \frac{TN}{TN+FP} \quad (4.12)$$

x) Cohen’s Kappa Score: Cohen's Kappa is a statistic that evaluates the agreement between predicted and actual classifications while accounting for the possibility of agreement occurring by chance. It considers observed agreement and expected agreement, providing a measure of agreement beyond what would be expected randomly. Equation (4.13) represents the formula of calculating CKS.

$$CKS = \frac{P(A)-P(E)}{1-P(E)} \quad (4.13)$$

Where:

P(A) is the relative observed agreement among raters or classifiers.

P(E) is the expected agreement that would occur by chance.

The formula calculates the Kappa Score by subtracting the expected agreement (chance agreement) from the observed agreement and then normalizing the result. The

resulting value ranges from -1 to 1. A CKS of 1 indicates perfect agreement between the raters or classifiers. A CKS of 0 suggests that the observed agreement is no better than what would be expected by chance alone. A CKS less than 0 indicates agreement worse than what would be expected by chance.

4.1.7 Forming Developer Team

In the bug-triaging process, the aim is to assemble a developer team with the most suitable skills and expertise to tackle a variety of software bugs efficiently. To achieve this, we employ a systematic approach wherein each type of developer is carefully selected based on the outcomes generated by the best classifier for their respective skill sets and qualifications. This strategic selection process ensures that we create a developer team composed of individuals who are not only highly capable but also well-matched to the specific characteristics and requirements of each bug. By categorizing developers according to their skills and expertise, we can effectively distribute bug resolution tasks among different team members. This approach optimizes the utilization of developer resources and ensures that each bug report is assigned to the most qualified individuals who are best equipped to address its unique challenges. As a result, our bug-triaging system promotes collaboration and specialization within the developer team, leading to improved bug resolution times and enhanced software quality. In essence, our approach goes beyond mere bug assignment; it facilitates the formation of developer teams that are tailored to the diverse nature of software bugs, ultimately leading to more effective bug resolution and a smoother development process.

4.2 Result Analysis

The result analysis section of our study delves into the comprehensive evaluation of our Bug Solving Developer Recommendation Model. In this phase, we take a meticulous approach to assess the performance and effectiveness of BSDRM in the context of developer recommendations for bug resolution. Our analysis begins by describing the key components and methodologies employed in BSDRM, highlighting the utilization of pre-trained BERT for sentence embedding and applying resampling techniques to balance the dataset. To rigorously evaluate the model, we use an array of diverse classifiers, including Decision Trees, Extra Trees, AdaBoost, Bagging Classifier, Gradient Boosting Classifier, Random Forest, K-Nearest Neighbors, Nearest Centroid, Bernoulli Naïve Bayes, Multinomial Naïve

Bayes, Complement Naïve Bayes, Gaussian Naïve Bayes, Support Vector Machine, Stochastic Gradient Descent, Logistic Regression, Perceptron, and Multi-Layer Perceptron. These classifiers are meticulously selected to identify and categorize different types of developers, forming the foundation of our developer recommendation system. Different performance metrics serve as essential benchmarks for assessing the effectiveness of our proposed BSDRM, which is also compared against traditional models to highlight its advancements and contributions to the field of developer recommendation for bug resolution.

4.2.1 Classification Results of BSDRM

This study merges all datasets to create a developer matrix, which is subsequently divided into training and testing sets. The training set undergoes further division into two parts: one part is used to train a sentence embedding model for generating a bag of developers' words, while the other part is dedicated to training a developer classifier comprising various classifiers such as DT, ET, AdC, BC, GB, KNN, NC, BNB, MNB, CoNB, GNB, LR, Pr, and MLP. The embedded model generates a vocabulary list when assessing new bug reports or testing instances. Subsequently, an unsupervised KNN finder matches the bag of developers' words with the testing vocabulary list to extract eligible developers. Using the developer classifier, these developers are then categorized into Experienced Developers, New Experienced Developers, and Fresh Graduates. These developer groups are subsequently combined to form a bug-fixing team. To evaluate the performance of BSDRM, we conducted a case study involving open-source projects, including Eclipse, Mozilla, and Netbeans. The data matrix is manually labeled with the respective developer categories (ED, NED, and FG), and individual classifiers are employed to classify developers accordingly. The experimentation and evaluation occur within the Google Colaboratory environment using Python, with the sci-kit learn library as a crucial tool. Table 4.2 provides a comprehensive summary of the experiment results obtained from each classifier, focusing on key performance metrics such as Accuracy, Precision, Recall, F1-Score, Hamming Loss, Jaccard Score, Matthews Correlation Coefficient, Area Under the Curve, Balanced Accuracy, and Cohen’s Kappa Score. This thorough evaluation offers insights into the effectiveness of BSDRM in accurately classifying developers and forming well-balanced bug-fixing teams for improved bug resolution outcomes.

Upon analyzing the comprehensive dataset presented in Table 4.2, it is evident that BC consistently demonstrates remarkable performance across various evaluation metrics,

showcasing its effectiveness as a reliable classifier for developer classification. However, it's worth noting that AUC, an essential metric for assessing classifier performance, notably excels by ET, signifying its proficiency in achieving a high area under the curve. Furthermore, RF and GBC excel in accuracy, boasting an impressive rate of 96.42% for accurately categorizing developers. Notably, DT, ET, and AdC consistently yield favorable results, with most of their scores surpassing the 90% mark, indicating their robustness in developer classification. Conversely, KNN, SGB, and MLP exhibit performance levels in the 80% range, indicating their moderate efficiency in this task. GNB and Pr, while not achieving as high accuracy as some other classifiers, deliver respectable outcomes, hovering around the 70% mark. In contrast, several classifiers in the experiment need to be more accurate in accurately identifying developers, emphasizing the importance of selecting the appropriate classifier for such a nuanced task. As a result, the superior performance of BC in classifying developer experience levels, as indicated by most evaluation metrics, establishes it as the standout choice for this critical task, ensuring the reliability of developer classification within the BSDRM model.

Table 4.2: Developer classification result

Classifier	Acc (%)	Prec (%)	Rec (%)	FS1 (%)	HL (%)	JS (%)	MC (%)	AUC (%)	BAC (%)	CKS (%)
DT	95.74	95.75	95.77	95.75	4.26	91.90	93.61	98.86	98.86	93.62
ET	94.89	94.95	94.96	94.95	5.12	90.49	92.34	**99.23**	94.96	92.34
AdC	68.82	74.42	69.81	67.17	31.18	53.59	56.81	95.35	69.80	53.52
BC	**96.59**	**96.62**	**96.56**	**96.59**	**3.41**	**93.42**	**94.89**	98.73	**96.56**	**94.88**
GB	96.42	96.45	96.41	96.43	3.58	93.15	94.63	98.85	96.41	94.63
RF	96.42	96.48	96.40	96.43	3.58	93.15	94.63	98.99	96.40	94.63
KNN	88.93	88.92	88.90	88.82	11.07	80.12	83.47	94.24	88.90	83.39
NC	35.96	35.97	36.52	34.03	64.04	20.75	5.62	48.80	36.52	4.98
BNB	47.02	55.79	47.88	42.45	52.98	27.60	27.20	66.25	47.89	21.40
MNB	45.51	47.51	47.28	38.80	53.49	26.32	26.12	68.34	47.28	20.54
CoNB	44.97	44.98	45.65	35.58	55.03	23.56	24.32	67.51	45.65	18.16
GNB	79.39	83.12	80.14	79.06	20.61	67.78	71.55	99.00	80.13	69.19
SVM	69.85	75.62	69.86	70.53	32.15	54.76	56.35	83.61	69.86	54.84

SGD	83.65	83.65	83.53	82.28	16.35	71.02	77.18	95.71	83.53	75.41
LR	56.86	60.65	54.77	56.95	46.14	40.26	34.53	65.70	54.77	32.62
Pr	78.02	79.83	77.53	77.45	22.99	64.76	68.02	86.78	77.52	66.97
MLP	87.56	88.08	87.46	87.09	33.44	78.86	82.97	92.71	88.46	81.35

4.2.2 Comparisons with Existing Works

Many previous studies have explored the domain of bug assignment, often focusing on assessing developers' capabilities by examining their past bug-solving achievements [91, 137, 65]. Some research even introduced a fresh approach by incorporating New Experienced Developers, a category encompassing individuals with intermediate expertise or diverse backgrounds, as well as Fresh Graduates into their bug-solving initiatives [138]. In our endeavor, we have adopted a notably comprehensive approach by including these developer categories alongside Experienced Developers, forming teams well-equipped to address emerging bug issues. This approach not only empowers NED and FG developers to participate actively in the bug triage process but also fosters their learning through engagement. Our innovative model, BSDRM, stands out by automatically assigning bug reports to the most suitable developers, effectively reducing the likelihood of reassigning identical bugs to the same developers. To evaluate BSDRM's superiority over established methodologies, we subject it to scrutiny based on three critical criteria:

C1 - Bug Fixing History Estimation: When assigning a new task, does BSDRM assess a developer's bug-fixing history?

In its bug assignment process, BSDRM considers the bug-fixing history of developers. It evaluates their past experiences and assigns them new tasks accordingly. This ensures that developers with relevant bug-fixing expertise are given tasks that align with their historical performance.

C2 - Developer Interest Consideration: Does BSDRM consider a developer's interest in the task allocation scheme?

BSDRM not only considers developers' historical bug-fixing records but also takes into account their interests and preferences when allocating tasks. This approach ensures that developers are assigned tasks that align with their skills and interests, increasing their motivation and effectiveness in bug resolution.

C3 - Diverse Developer Team Formation: Does the assigned team consist of developers from all experience levels?

BSDRM creates bug-fixing teams that include developers from different experience levels, namely Experienced Developers, Newly Experienced Developers, and Fresh Graduates. This approach ensures that bug assignments are distributed across a diverse range of developers, optimizing team composition and enabling knowledge transfer among team members.

Table 4.3: Comparison of BSDRM with traditional models

	Core Concepts	**C1**	**C2**	**C3**
BugFixer [46]	Text exploration	Yes	No	No
Jeong et al. [91]	Tossing graph	Yes	No	No
Yadav et al. [61]	Tossing length	Yes	Yes	No
CosTriage [99]	Cost aware ranking	Yes	No	No
DEVELECT [97]	Source code-based approach	Yes	No	No
Zhang et al. [48]	Social network based	Yes	Yes	No
DRETOM [47]	Topic model based	Yes	Yes	No
Shokripour et al. [98]	Time based approach	Yes	No	No
Khatun et al. [137]	Time based approach	Yes	No	No
TEAN [138]	LDA	Yes	Yes	No
BSDRM (Proposed Model)		Yes	Yes	Yes

We conduct a comprehensive comparison, presented in Table 4.3, to underpin our analysis. Our evaluation reveals that BSDRM, like previous models, excels at identifying experts based on their bug resolution track record (effectively addressing C1). Interestingly, BSDRM aligns with models such as [138, 47, 99, 48], which take into account developers' enthusiasm for tackling new bugs, thus efficiently handling C2. Notably, BSDRM, in conjunction with TEAN, ensures the formation of diverse teams comprising various developer profiles [11]. However, a significant limitation of TEAN is its inability to effectively accommodate mid-level developers and experts from different domains across multiple organizations (C3). In contrast, BSDRM adeptly accommodates three developer categories (ED, NED, and FG), resulting in a robust and balanced team while considering

their levels of experience. Therefore, BSDRM effectively addresses C3, as supported by the data in Table 4.3. It is essential to acknowledge that BSDRM may encounter challenges in consistently and accurately assessing the proportions of ED, NED, and FG developers in all cases.

4.3 Summary

Bug triage involves the allocation of bugs to developers based on their past experiences, a crucial process for ensuring a balanced workload among developers. We introduce the Bug Solving Developer Recommendation Model, a machine learning-based bug-triaging approach to address this challenge. Our approach begins by collecting and combining various datasets, which are then divided into training and testing sets. Subsequently, we construct a sentence-embedded model using the training set, generating a collection of developer-related terms. The test set is transformed into a vocabulary list using this embedded model. BSDRM employs the K-Nearest Neighbor algorithm to suggest eligible developers by comparing the developer term collection with the bug report vocabulary list. These recommended developers are subsequently classified, including experienced, newly experienced, and fresh graduate developers, using a diverse set of classifiers such as Decision Trees, Extra Trees, AdaBoost, Bagging Classifier, Gradient Boosting Classifier, Random Forest, K-Nearest Neighbors, Nearest Centroid, Bernoulli Naïve Bayes, Multinomial Naïve Bayes, Complement Naïve Bayes, Gaussian Naïve Bayes, Support Vector Machine, Stochastic Gradient Descent, Logistic Regression, Perceptron, and Multi-Layer Perceptron. Notably, BC demonstrates remarkable accuracy at 96.59% in classifying developers with varying experience levels. Based on these outcomes, BSDRM recommends assembling developer teams to address testing bugs.

Chapter 5

Task Allocation and Load Balancing

Bug triage is the structured process of prioritizing and resolving reported software bugs based on factors like severity, urgency, and impact on functionality. It ensures that resources are allocated efficiently and that critical issues receive immediate attention, ultimately leading to the identification of underlying software problems. However, several challenges arise if bugs are not appropriately distributed among developers. Experienced developers may become overwhelmed with critical bugs, while mid-level developers from different fields and fresh graduates may need suitable assignments to gain valuable experience. This situation can hinder career progression, reduce job satisfaction and productivity, and delay bug-fixing.

Conversely, skilled developers' job satisfaction and productivity may suffer if skilled developers are consistently overloaded. Sometimes, medium or newly experienced developers might attempt to resolve bugs without proper investigation, leading to increased costs and time to rectify the issues. Additionally, fresh graduates may need more opportunities to learn bug-solving skills, potentially resulting in high professional turnover rates. Assigning complex or critical bugs to inexperienced developers without proper guidance can have detrimental effects on software performance and user confidence.

Several previous works have aimed to identify and recommend developers for bug resolution, but a common shortcoming is the lack of workload distribution considerations among different developer types. We introduce a task allocation model called Developer Scheduler to address this issue. DevSched assigns bugs to developers based on minimum requirements, competencies, and workload considerations, ensuring that various bug types are appropriately distributed among experienced, medium-experience, and fresh graduate developers. The model follows a series of steps to perform task allocation activities.

5.1 Overview of DevSched

In this section, we present an innovative automated bug assignment solution named Developer Scheduler (DevSched), meticulously designed to tackle bug-fixing challenges and enhance the efficiency of developer workloads. As emphasized earlier, our approach places considerable importance on assessing developer proficiency through their previous bug-solving experiences [17]. Additionally, we recognize that factors like team changes and participation in multiple projects can influence the success of bug assignments. DevSched employs a sophisticated task-scheduling algorithm that distributes bugs to developers, considering their levels of expertise and ongoing project commitments. The intricate details of this task scheduling methodology are illustrated in Figure 5.1, providing a comprehensive overview of its functionality and impact on bug assignment optimization.

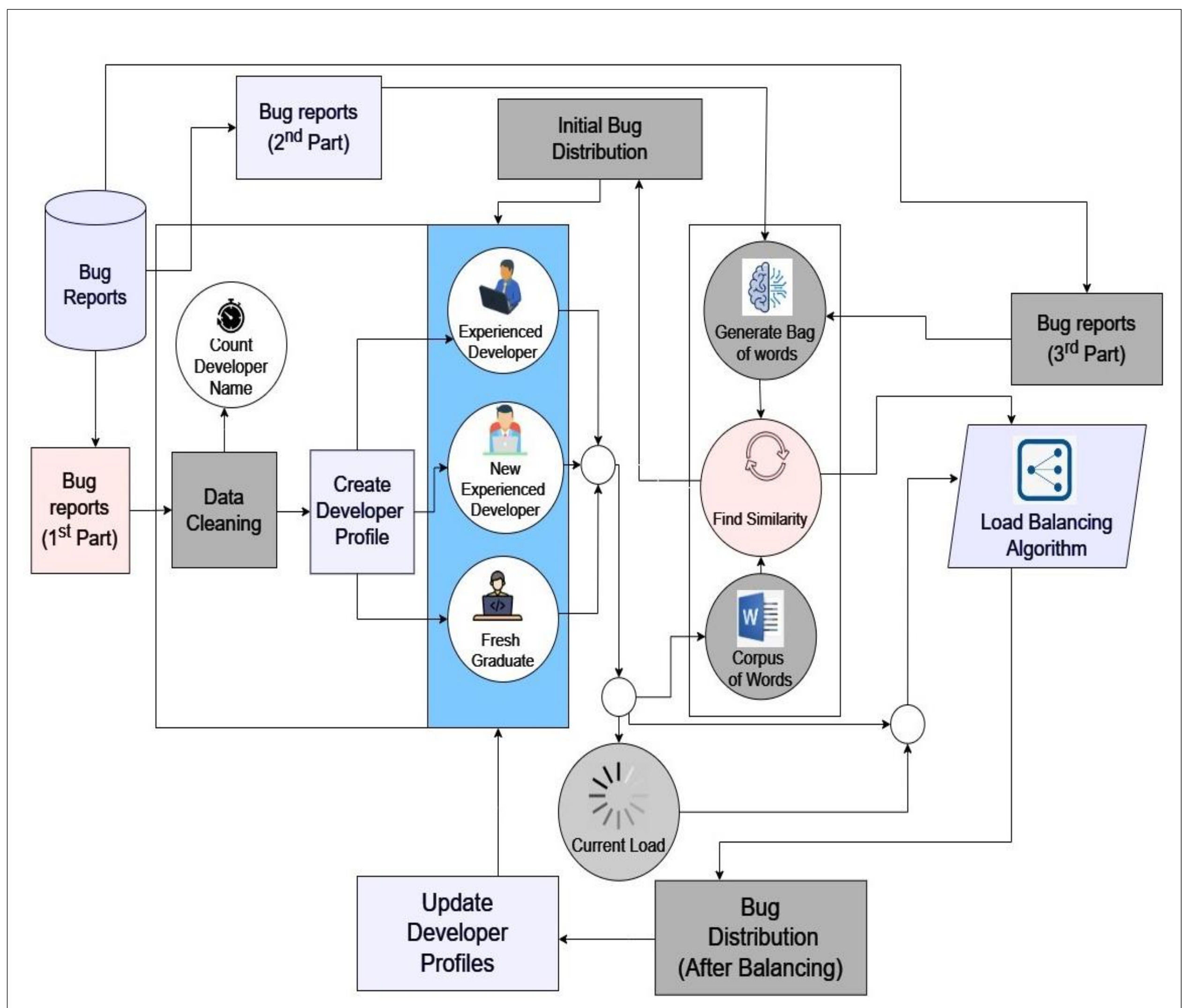


Figure 5.1: DevSched: Task allocation and load balancing model

5.1.1 Data Pre-processing

To streamline the bug assignment procedure, DevSched begins by partitioning the experimental datasets into three distinct segments, each serving a specific role. The initial segment is dedicated to the creation of diverse developer profile ratings. The second portion of the datasets is reserved for the preliminary bug assignment to individual developers before any workload balancing measures are applied. Finally, the last segment of these datasets plays a crucial role in workload equalization among developers and the continuous updating of their statuses based on their evolving competencies. This structured approach ensures a systematic and efficient bug assignment process.

5.1.2 Developer Profile Rating Calculation

The initial phase of bug report handling involves collecting and categorizing bug reports into various tiers. This multifaceted process consists of two fundamental steps: tallying developer names and constructing a word corpus. Firstly, the model eliminates common stop words from each developer's bug summaries and descriptions. Employing regular expressions, unique developer names are extracted from the "Assignee" and "Single Developer" fields. Subsequently, each developer's record incorporates details regarding the priority (ranging from P1 to P5) and severity (comprising Enhancement (E_h), Trivial (T_r), Minor (M_n), Normal (N_l), Major (M_j), Critical (C_c), Blocker (B_l)) of the bugs they have addressed.

On the other hand, a word corpus is meticulously crafted from the bug descriptions and summaries by applying lemmatization techniques. This corpus enables the sequential identification of each developer instance. Furthermore, a rating procedure (R_t) is implemented for each developer, taking into account the priority (P_t) and severity (S_v) of the resolved bugs. These factors are computed using the following Equations (5.1) - (5.3):

$$P_t = P_1 \times 5 + P_2 \times 4 + P_3 \times 3 + P_4 \times 2 + P_5 \times 1 \quad (5.1)$$

$$S_v = (E_h + T_r) \times 1 + (M_n + N_l + M_J) \times 2 + (C_r + B_l) \times 3 \quad (5.2)$$

From Eq. 5.1 and 5.2:

$$R_t = P_t + S_v \quad (5.3)$$

Developer profiles are intricately constructed, focusing on their aptitude for resolving bugs. Developers are systematically classified into three discernible categories: Experienced Developers, New Experience Developers, and Fresh Graduates, contingent on an array of diverse criteria. The formulation of these profiles hinges on the developer's rating (R_t), meticulously computed through the utilization of P_t and S_v.

5.1.3 Load Creation

During the workload creation phase, the datasets' second segments assume a central role in the generation of workloads tailored to different developer categories. This phase revolves around allocating bugs to developers best suited for the task, achieved through thoroughly analyzing bug attributes and their alignment with corresponding developer profiles. The workload creation process unfolds through a sequence of discrete steps, each contributing to the efficient assignment of tasks. Let's delve into these steps in more profound detail to gain a comprehensive grasp of this process.

5.1.3.1 Data Transformation

In order to enable a comprehensive comparison of diverse bugs and evaluate their similarities, each bug undergoes a rigorous transformation process, ultimately converting them into sets of vectors. This transformation is a pivotal step that involves applying term frequency (TF) and inverse document frequency (IDF) techniques, which play a crucial role in fine-tuning the importance of specific feature words within the vector space model. Delving deeper into these concepts will provide a more nuanced understanding of their significance in this context:

i) Term Frequency: TF is a crucial metric that measures the frequency of a specific term or word within a given bug report or document. It quantifies how often a particular term appears, assigning a weight to each term based on its frequency within that document. The number of occurrences $n_{i,j}$ of a particular word t_i divided by the total number of words in the d_j document is defined as TF. The formula is expressed as Equation (5.4). This approach ensures that terms occurring more frequently within a bug report carry higher weights, thereby making them more influential when constructing the vector representation of the bug report.

$$tf_{i,j}=\frac{n_{i,j}}{\sum_k n_{k,j}} \tag{5.4}$$

Where, $\sum_k n_{k,j}$, is the sum of the occurrences of all words in the d_j .

ii) Inverse Document Frequency: IDF complements the TF metric by assessing the significance of a term across the entire dataset or collection of bug reports. IDF considers the rarity or uniqueness of a term within the corpus of bug reports. Terms prevalent across numerous bug reports receive lower IDF scores as they are considered less distinctive. Conversely, terms that appear in only a limited number of bug reports receive higher IDF scores, signifying their potential importance in distinguishing and characterizing bugs

effectively. This combination of TF and IDF techniques is pivotal in crafting meaningful vector representations of bug reports for similarity assessment and assignment purposes. The logarithm of the total number of documents |D| divided by the number of documents $|\{j : t_i \in d_j\}|$ containing a specific word t_i is defined as IDF. In cases where the word is not prevalent in the corpus, it could lead to a division by zero. To avoid this, a common practice is to add 1 to the denominator, resulting in the following formula Equation (5.5):

$$idf_i = \log \frac{|D|}{|\{j: t_i \in d_j|\}} \qquad (5.5)$$

This modified formula ensures that even rare terms have a non-zero IDF score.

We generate a vector space model for bugs by utilizing the TF-IDF measure, which involves considering both Term Frequency and Inverse Document Frequency to determine word weights. Furthermore, we apply lemmatization techniques to derive a word corpus from the developer profiles.

5.1.3.2 Bug Distribution

This section outlines a procedure for allocating bugs to developers by assessing the similarity between bug properties and a word corpus extracted from developer profiles. To measure this similarity, we employ cosine similarity, represented by Equation (5.6), to match and assign developers to specific bugs.

$$\cos\theta = \frac{A.B}{\|A\| \, \|B\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}} \qquad (5.6)$$

5.1.4 Load Balancing

Following the load creation phase, the next critical step involves balancing developers' workloads using our proposed load balancing method. Initially, we establish an average threshold considering the total number of developers and their existing workloads. Bug severity is crucial in determining the allocation of bugs among experienced developers, newly experienced developers, and fresh graduates. The algorithm detailing load balancing is presented in Algorithm 5.1.The three instances of the loadBalancer function in the DevSched function handle bugs of different severity levels (high, medium, and low) and allocate them to suitable developers of different types (ED, NED, FG) based on their current workloads and the calculated average threshold. If the new bug severity is high, first assign the experienced developer based on the average threshold. If all experienced developers are overloaded, then give priority to new-experienced developers. If all new, experienced developers are

overloaded, they are given to fresh graduate developers. If the new bug severity is medium, prioritize NED, FG, and ED, respectively. If the new bug severity is low, prioritize FG, NED, and ED, respectively. This modular approach enhances the readability, maintainability, and reusability of the code by encapsulating specific functionality within well-defined functions.

Once we've determined the average threshold and assessed bug severity, we examine each developer's current workload and allocate bugs to them according to their respective categories (ED, NED, FG). For instance, if Hasan, an ED, has 9 bugs assigned to him, Rahim, another ED, has 3 bugs, and Karim, an FG, has 2 bugs, the average threshold is calculated as (9+3+2)/3 = 4.67. In cases where the average threshold surpasses a developer's current workload, any incoming bugs are assigned to other developers in the same category who possess the lightest workloads. If all EDs are occupied, the subsequent bugs are assigned to NEDs or FGs following the same procedure. In the earlier example, Hasan's current workload consists of 9 bugs, exceeding the average threshold of 4.67. Consequently, new bug reports are assigned to Rahim or other available EDs with lower workloads. If no EDs are available, the next set of bugs is assigned to developers in the subsequent category. For instance, if there are no EDs or NEDs available, the bugs are allocated to FGs. As elucidated in the study, the calculation of the average threshold involves an assessment of each developer's current workload, thereby encapsulating the impact of newly assigned bugs. For example, if the average threshold is initially determined as 4.67 and a new bug is subsequently assigned to Rahim, the workload distribution is updated to include Rahim's additional assignment. Consequently, the average threshold is recalculated, taking into account the revised workload scenario, ensuring that it accurately represents the workload status among developers in the same category. This adaptive mechanism ensures that the bug allocation system remains responsive to changes in workload distribution, optimizing the efficiency of bug assignment while considering the evolving circumstances of each developer's responsibilities.

As developer profiles evolve over time, average threshold values are recalculated, and bug assignments are adjusted accordingly. This iterative process is repeated for each category until all bugs are resolved. In the final phase, we generate several graphs to visualize changes in standard deviations of workloads across different developers. These graphs provide insights into workload variations over time, both before and after implementing the load- balancing method.

Algorithm 5.1 Load Balancer Algorithm

Input: D is a set of developers, L_D is the current load of each developer in D, L_A be the average threshold value for ED, NED and FG, and S_v be the severity of the new bug report. T_1, T_2 and T_3 indicate different types of developers.

Output: Bug Assign and Balance Bugs among ED, NED, and FG.

def loadBalancer (S_v, T_1, T_2, T_3):

  if Severity is S_v **then**

    for each $T_1 \in D$ **do**

    if $L_D(d) < L_A$ **then**

      Assign the bug to suitable T_1 **break**

    end

    if none is assigned **then**

      for each $T_2 \in D$ **do**

        if $L_D(d) < L_A$ **then**

          Assign the bug to suitable T_2 **break**

        end

      end

      for $T_3 \in D$ **do**

        if $L_D(d) < L_A$ **then**

          Assign the bug to suitable T_3 **break**

        end

      end

    end

  End

End

Function DevSched ():

  for each $d \in D$ **do**

    $L_D(d) \leftarrow$ current load of developer d

  end

  $L_A \leftarrow \left\lceil \frac{\sum_{d \in D} L_D(d)}{|D|} \right\rceil$

  loadBalancer (high, ED, NED, FG)

  loadBalancer (medium, NED, FG, ED)

  loadBalancer (low, NED, FG, ED)

5.1.5 Update Developer Profile

The bug distribution outlined above serves as a basis for updating the profiles of Experienced Developers, New Experienced Developers, and Fresh Graduates in several ways:

Experienced Developers: EDs' profiles are continuously updated based on their bug-solving capabilities, mainly focusing on their efficiency in resolving bugs within a specified timeframe. Additional points are awarded to EDs who successfully handle more complex and critical bugs, reflecting their expertise in addressing challenging issues. Conversely, if EDs encounter difficulties in resolving bugs within the expected timeframes, their profiles undergo fewer updates. In some cases, it may be inferred that these developers are less experienced in handling specific types of bugs.

New Experienced Developers: NED profiles are updated based on their success rate in addressing medium-level bugs. NEDs, especially those transitioning from other domains, often begin by tackling easier bugs to familiarize themselves with bug-fixing criteria. If NEDs consistently demonstrate the ability to resolve critical and medium-level bugs over a certain period successfully, they are promoted to the status of EDs. Similar to EDs, NED profiles are not frequently updated when they face challenges in problem-solving. However, persistent difficulties in fixing specific issues may lead to weaknesses being reflected in their profiles.

Fresh Graduates: FG profiles evolve based on their performance in resolving low-level bugs. Although FGs primarily work on low-level bugs, they may occasionally assist NEDs/EDs in handling mid-level and critical problems under their guidance. Promotions from FG to NED are contingent on their consistent proficiency in handling low-level bugs over a specified timeframe. If FGs encounter difficulties resolving multiple low-level bugs, their profiles receive fewer updates. Continuous struggles may result in their profiles highlighting areas for improvement in addressing this particular task.

In summary, updating developer profiles based on their performance serves as a valuable mechanism for refining bug allocation and enhancing system efficiency. It ensures that developers are assigned tasks commensurate with their skills and experience, optimizing the bug resolution process.

5.2 Experimental Setting

In this section, we provide a concise overview of the essential materials required for the upcoming procedures.

5.2.1 Data Description

We obtained datasets from Bugzilla, specifically from Eclipse, Mozilla, and NetBeans, which collectively comprise extensive bug reports spanning multiple years. The Eclipse dataset contains 1, 37,147 bug reports from 2001 to 2020, the Mozilla dataset includes 13, 2261 bug reports ranging from 1999 to 2020, and the NetBeans dataset encompasses 44,149 bug reports recorded between 2001 and 2017. These datasets encompass various aspects of software projects, including source code, commit logs, and bug-related data. Notable features within these datasets include bug severity, priority, fix status, platform or hardware details, assignee, single developer involvement, first assignee, developer comments, bug summary (title), and detailed bug descriptions. It's important to note that the developers in these datasets have not been categorized or labeled based on their expertise. To facilitate our research, we divided these datasets into three distinct segments, each serving a specific purpose: profile creation, load assignment, and load balancing, as outlined in Table 5.1. This division enables us to manipulate and analyze the data effectively to address the challenges of bug assignment and workload optimization.

Table 5.1: Date ranging

Dataset	Profile Creation	Load Creation	Load Balancing
Eclipse	09-02-2017 to 14-02-2018	15-02-2018 to 13-03-2019	15-03-2019 to 12-06-2020
Mozilla	24-07-2017 to 26-08-2018	27-08-2018 to 01-07-2019	02-07-2019 to 05-12-2020
NetBeans	17-02-2009 to 08-02-2011	09-02-2011 to 09-04-2013	10-04-2013 to 22-12-2017

5.2.2 Environment Setup

For the experimental implementation of DevSched, we conducted our research within the Google Colab environment, leveraging the Python programming language. The computational resources utilized for this experiment included a personal computer equipped with 8 GB of RAM, a 3rd generation Intel Core i5 processor operating at 2.8 GHz, and the Windows 10 operating system. Additionally, we employed the Natural Language Toolkit (NLTK) as a critical tool for the implementation of DevSched, enabling us to carry out our bug assignment and workload optimization tasks effectively.

5.3 Result Analysis

This section details our comprehensive approach to using DevSched to manage the assignment of bugs to developers and optimize their workloads across three distinct datasets: Eclipse, Mozilla, and NetBeans. Each dataset consists of 10,000 bugs in its initial part, serving as the foundation for the developer profile creation process (as seen in Table 5.1). Here, we elucidate the intricate steps involved in this multifaceted process, delineating how it unfolds across the different segments of the datasets.

a) Profile Creation Phase

In the profile creation phase, we utilize the initial segments of the three datasets, each containing 10,000 bugs. These bugs serve as the basis for creating developer profiles. We commence by identifying the number of individual developers and subsequently establish a primary threshold crucial for the selection of suitable developers within each dataset, as delineated in Table 5.2. Additionally, we define distinct thresholds for Experienced Developers, New Experience Developers, and Fresh Graduates in each dataset. Thus, the first parts are instrumental in determining these thresholds for various developer categories across datasets.

b) Load Creation Phase

Following the profile creation, the second part of these datasets are employed to generate new workloads for EDs, NEDs, and FGs. Here, the bugs are transformed into vectors using the TF-IDF method, while Lemmatization techniques are applied to convert developer profiles into a corpus of words. The cosine similarity method plays a pivotal role in assigning the most suitable developers to address different bug types. The second segments are pivotal in creating initial workloads, taking into account the primary thresholds for EDs, NEDs, and FGs in their respective datasets.

c) Load Balancing Phase

The last segments of these datasets are dedicated to assigning and balancing bugs among developers using the load balancing algorithm (Algorithm 5.1). In this phase, the DevSched method employs a common overall threshold to categorize bugs and assigns them to EDs, NEDs, and FGs accordingly. The load-balancing process continues iteratively until all bugs find suitable developers.

Table 5.2: Load thresholds of different types of developer

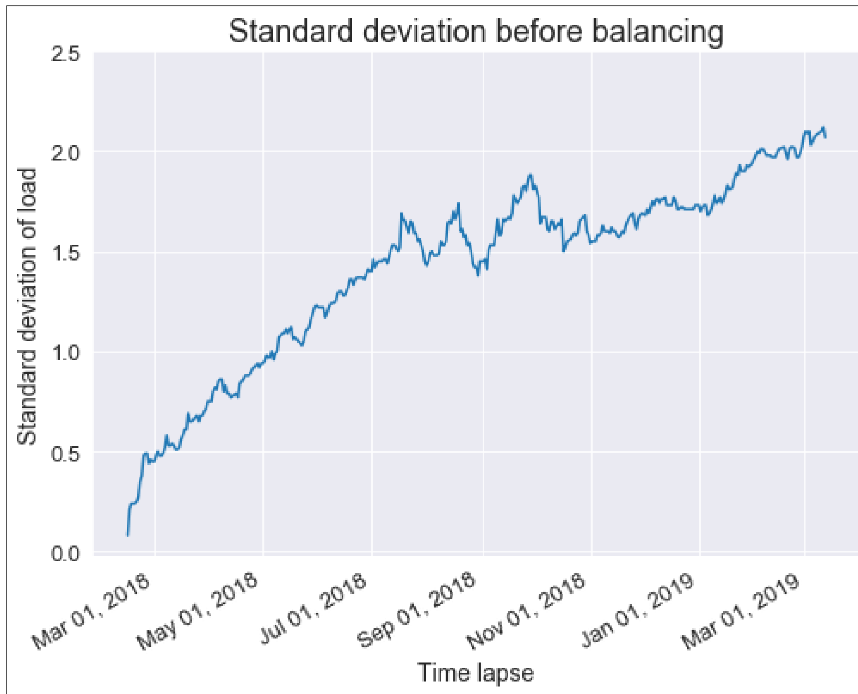
Data	Number of Bugs	Number of Devs	Number of Selected Devs.	Threshold for Selected Devs.	The Threshold for Devs. Class				
					FG		NED		ED
Eclipse	10000	1097	553	>10	> 10	<75	>75	<150	>150
Mozilla	10000	2332	463	>80	>80	< 200	> 200	< 400	>400
Netbeans	10000	1879	492	>20	>20	< 200	> 200	< 400	>400

5.3.1 Results of Eclipse Dataset

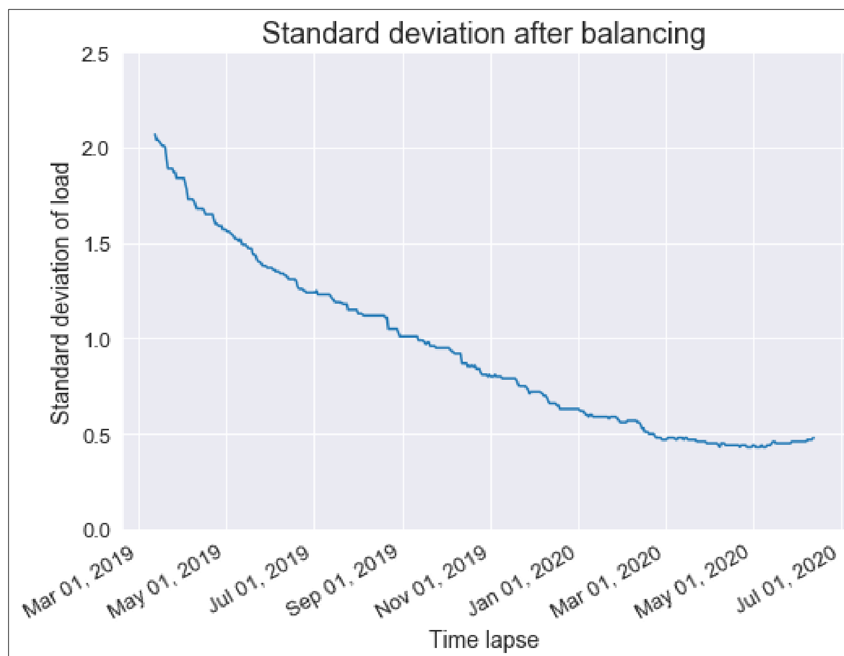
Within the Eclipse dataset, DevSched is harnessed to harmonize the workloads of individual developers. We meticulously assess the standard deviations of bugs both before and after the integration of the load-balancing mechanism. Initially, in Figure 5.2a, the standard deviations exhibit subtle fluctuations as different bugs are assigned by their predefined thresholds. However, it becomes evident that this allocation needs more proper equilibrium.

Upon the integration of the load-balancing methodology, a marked transformation occurs in the appropriateness of developers' workloads. This transformation is discerned by the conspicuous reduction in the standard deviation, as elucidated in Figure 5.2b. The efficacy of this bug distribution is further substantiated by the structured representation of bug allocation in Table 5.3. Following the creation of developer profiles, we identified 130 developers in the Experienced Developer category, 73 in the New Experience Developer category, and 350 in the Fresh Graduate category. Subsequently, the load creation process commences, culminating in the discovery of 162 EDs, 122 NEDs, and 269 FGs, respectively.

It is noteworthy that during the load creation phase, we discern a relatively consistent increment in the standard deviation, eventually reaching 2.07. However, with the judicious application of the load-balancing methodology among diverse developers, we unearth 261 EDs, 177 NEDs, and 115 FGs. In this scenario, an incremental decrement is observed in the standard deviation of developers' workloads, ultimately culminating in a remarkable reduction to 0.48.



(a)



(b)

Figure 5.2: The standard deviation curve depicts how the workload of developers has varied over time for Eclipse dataset

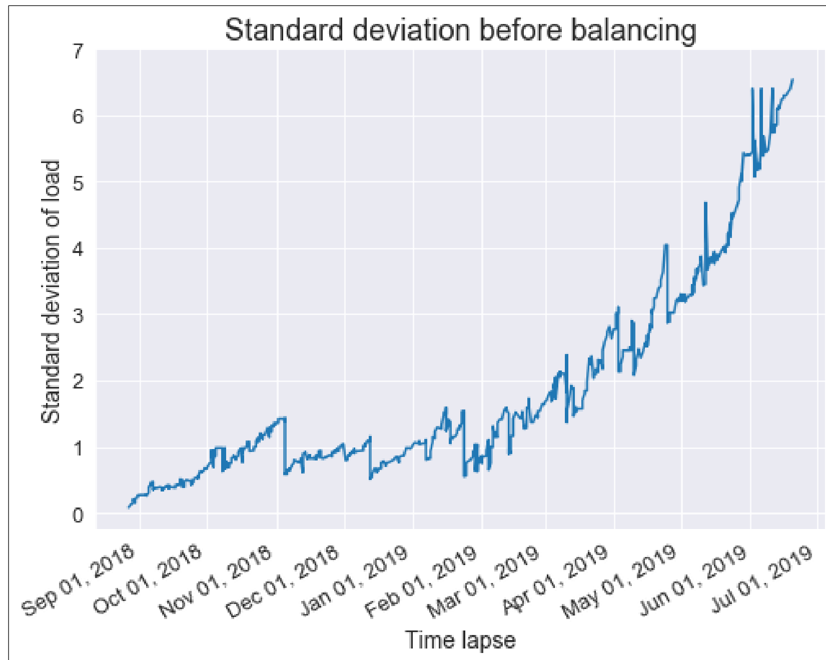
Table 5.3: The distribution of developer by implementing DevSched

Eclipse Dataset			
	Profile Creation	**Assign Load**	**Load Balancing**
ED	130	162	261
NED	73	122	177
FG	350	269	115
Mozilla Dataset			
	Profile Creation	**Assign Load**	**Load Balancing**
ED	167	178	263
NED	109	129	102
FG	187	156	98
NetBeans Dataset			
	Profile Creation	**Assign Load**	**Load Balancing**
ED	98	116	166
NED	63	66	59
FG	331	310	267

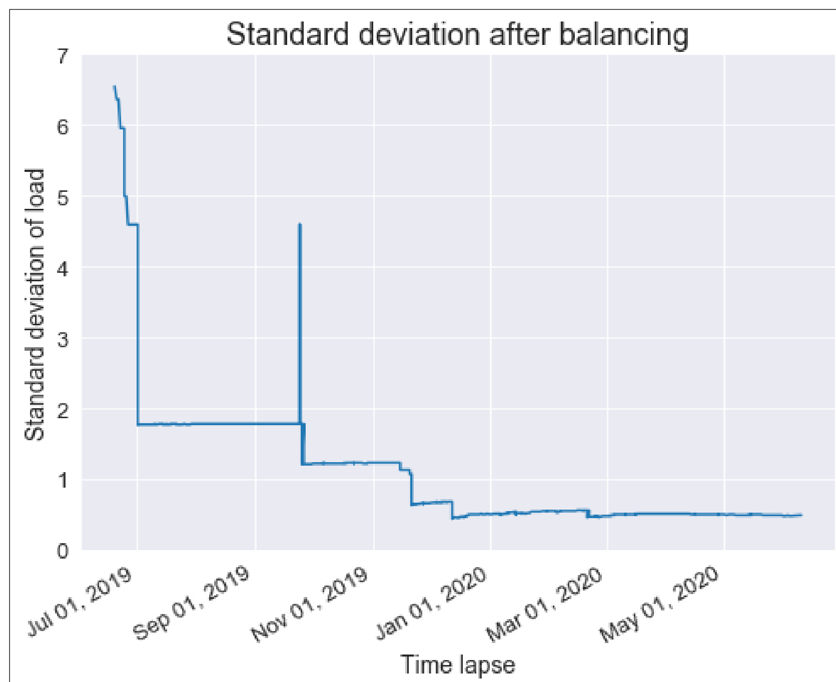
5.3.2 Results of Mozilla Dataset

Devsched is put into action within the Mozilla dataset, facilitating the equitable distribution of workloads among individual developers, as outlined in Table 5.3. In consonance with other datasets, the initial segment of this dataset comprises 167 Experienced Developers, 109 New Experience Developers, and 187 Fresh Graduates. This segment plays a pivotal role in establishing the fundamental thresholds for both the primary selection process and the categorization of developers. Subsequently, we meticulously orchestrate the allocation of workloads among diverse developers, adhering to these categorical thresholds, resulting in the emergence of 178 EDs, 129 NEDs, and 156 FGs.

The standard deviations of the workload distribution are diligently computed and presented in Figure 5.3a. In this context, the standard deviations exhibit fluctuations, culminating in an overall value of 6.54. Upon the implementation of the proposed load balancing mechanism, a discernible transformation is observed in the standard deviation curve, as depicted in Figure 5.3b. This transformation is indicative of a reduction in the standard deviation to a value of 0.49.



(a)



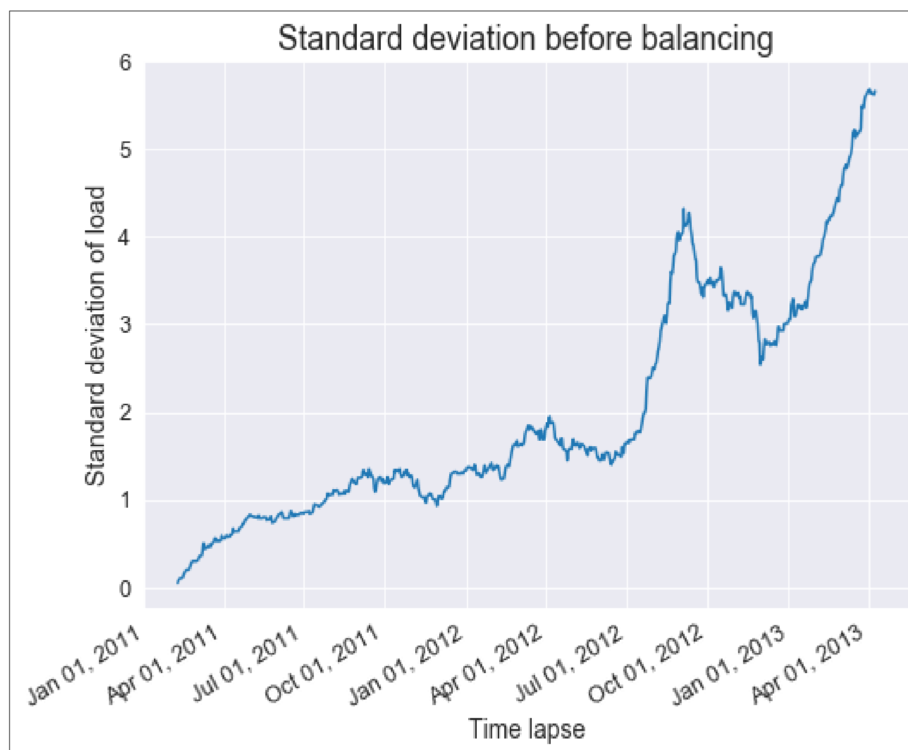
(b)

Figure 5.3: The standard deviation curve depicts how the workload of developers has varied over time for Mozilla dataset

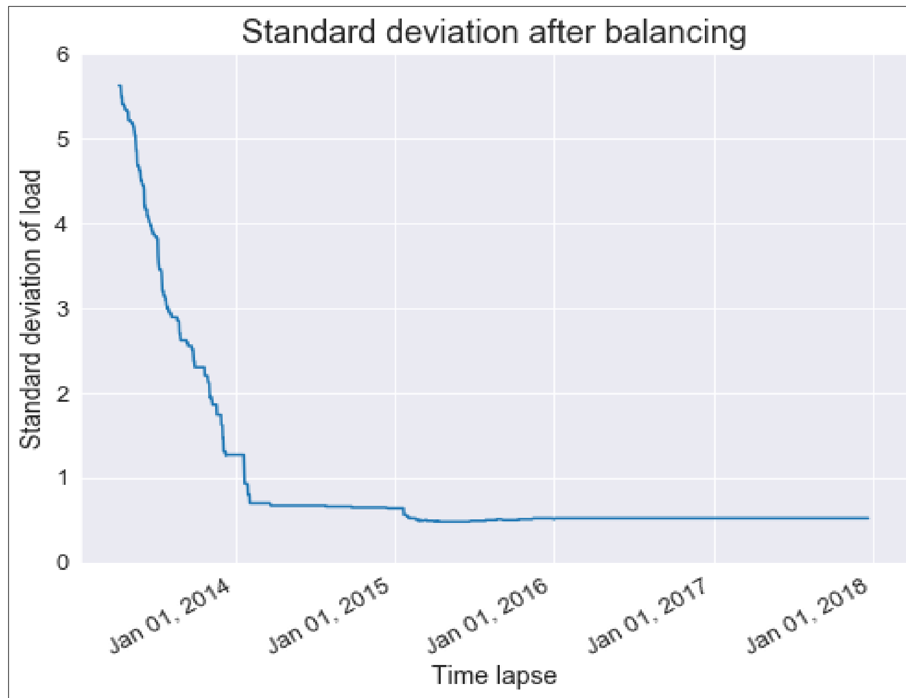
5.3.3 Results of NetBeans Dataset

Within the confines of the NetBeans dataset, DevSched takes center stage, orchestrating the harmonization of workloads among diverse developers through the judicious implementation of a load-balancing mechanism. In Figure 4a, a visual representation is presented, showcasing the standard deviations of workloads, which exhibit fluctuations in the absence of the load balancing method. However, with the strategic deployment of the load balancing approach, illustrated in Figure 5.4b, a notable transformation unfolds. This transformation manifests as a reduction in standard deviations, signifying the apt distribution of workloads among developers.

During the initial phase of profile creation, the NetBeans dataset reveals the presence of 98 Experienced Developers, 63 New Experience Developers, and 331 Fresh Graduates, as delineated in Table 5.3. Subsequently, as the primary workload is assigned in alignment with predefined thresholds, 116 EDs, 66 NEDs, and 310 FGs are entrusted with resolving various bugs. In this scenario, the overall standard deviation associated with the distribution of bugs among developers stands at 5.66. Following this phase, the proposed load-balancing methodology is activated, leading to the allocation of 166 EDs, 59 NEDs, and 267 FGs to address an array of bug types. This transformation leads to substantial improvements in the distribution of bugs among the various categories of developers, as evidenced by the diminished standard deviation, which culminates at 0.52.



(a)



(b)

Figure 5.4: The standard deviation curve depicts how the workload of developers has varied over time for the NetBeans dataset.

5.3.4 Comparative Analysis among Datasets

In the course of this research endeavor, a detailed examination of standard deviation values was carried out across various datasets, which can be shown in Table 5.4. Notably, the Eclipse dataset emerged with the lowest standard deviation, signifying a more balanced distribution of workloads among developers. Conversely, the NetBeans dataset exhibited a relatively higher standard deviation initially. However, the implementation of DevSched's load-balancing mechanism swiftly and effectively mitigated this discrepancy, rendering the workload distribution among individual developers more equitable. Conversely, the Mozilla dataset presented the maximum standard deviation, highlighting a broader variability in workload distribution among developers.

Nevertheless, DevSched, through its adept load-balancing strategies, substantially reduced this standard deviation. This observation further indicates that developers within the Mozilla dataset efficiently resolved bugs in a relatively shorter timeframe. Furthermore, it is worth noting that the Eclipse dataset demonstrated a higher degree of compatibility with the implementation of DevSched compared to the Mozilla and NetBeans datasets, underscoring the suitability and efficacy of DevSched in diverse contexts.

Table 5.4 Comparative analysis in terms of standard deviation

Dataset	Standard Deviation	
	Before Load Balancing	**After Load Balancing**
Eclipse	2.07	0.48
Mozilla	6.54	0.49
NetBeans	5.66	0.52

5.3.5 Comparisons with Existing Works

Numerous research efforts have delved into addressing the intricate challenges of bug triage. A wide array of computational techniques, spanning text categorization, graph-based approaches, cost-aware strategies, source-based methodologies, machine learning, deep learning, and hybrid methods, have been harnessed to facilitate the effective allocation of bugs to the most suitable developers. Historically, many prior studies predominantly leaned toward designating experienced developers as the go-to candidates for bug resolution. However, modern organizations have evolved, recognizing the importance of diverse talent pools that encompass not only seasoned developers but also individuals with varying skill levels, fresh graduates, and experts from different domains. Regrettably, these earlier approaches often failed to provide equitable solutions for distributing tasks among this heterogeneous workforce [46, 54]. The ramifications of neglecting to assign bugs to newly experienced developers and fresh graduates are profound. Such oversight denies them the opportunity to accumulate valuable experience, hindering their prospects for promotion to roles as newly experienced developers or experienced developers. The consequence is a palpable decline in job satisfaction and a loss of skilled personnel from the profession.

Recent research endeavors have started to acknowledge the significance of newly experienced developers, yet there remains a dearth of comprehensive characterizations for this category of developers [77, 79, 85]. It is imperative to rectify this situation by instituting a fair and efficient workload distribution system that accounts for developers of varying experience levels. Unfortunately, prior research efforts have largely omitted load distribution considerations, further underscoring the necessity for a novel approach [46, 78, 82, 88, 138].

In response to these challenges, we introduce DevSched, a novel framework that classifies developers into three distinct categories: Experienced Developers, Newly

Experienced Developers, and Fresh Graduates. DevSched employs a meticulously designed threshold system to allocate a predetermined number of bugs to each of these categories during the initial phase. Subsequently, it leverages a load-balancing methodology to ensure that the workloads of developers are more equitably distributed. This process is validated through empirical experimentation across different datasets, where DevSched consistently demonstrates its efficacy in reducing standard deviations of workload distributions. Table 5.5 depicts the comparisons of our proposed DevSched model with existing works.

Table 5.5 Comparisons with existing works

Reference	Differences
[77], [79], [85], [46], [78], [82], [88]	They consider experienced developers but do not consider new developers and load balancing.
[138]	They include new developers but exclude load balancing from consideration.
DevSched (Proposed Model)	***Consider both different types of developers and load balancing.***

In summary, the findings underscore the value of DevSched as a powerful tool for achieving a more balanced workload distribution among developers during bug allocation processes. This approach not only enhances software development efficiency but also ensures that developers of diverse backgrounds are assigned tasks within manageable workloads, fostering a more inclusive and effective development ecosystem.

5.4 Summary

Bug triage is a crucial element in software development, involving categorizing and prioritizing reported bugs. Effective bug triage ensures the efficient allocation of resources and helps uncover underlying issues within the software. However, when bugs are not appropriately distributed among different types of developers, it can lead to delays, errors, reduced capabilities, and lower job satisfaction. While previous research has proposed methods for recommending developers to fix bugs, they often need to pay more attention to proper workload distribution and balancing among developers. This study aims to introduce a task allocation and load balancing model, Developer Scheduler (DevSched), designed to distribute unassigned bugs among different types of developers efficiently. DevSched creates developer profiles based on existing bug reports and converts new bug reports into vectors.

Developer profiles are transformed into a corpus of words, and bugs are assigned to developers by comparing the cosine similarity between bug vectors and developer corpora. DevSched dynamically updates developer workloads and adjusts their ratings based on performance. To assess its effectiveness, Eclipse, Mozilla, and NetBeans bug reports are used to evaluate individual developers' performance in bug triaging.

The results demonstrate that DevSched assigns and balances bugs among different types of developers more efficiently. After applying the proposed load balancing model, standard deviations decrease rapidly compared to normal bug distributions across different datasets. This process is iterated for each bug, ensuring efficient resource allocation and critical issue resolution. As a result, the lowest standard deviation is achieved at 0.48 for the Eclipse dataset. The implications of DevSched's bug assignment based on defined thresholds, skills, and workloads include improved efficiency, reduced delays, and enhanced job satisfaction among developers.

Chapter 6

Discussion and Conclusion

The need for an automatic bug-triaging and load-balancing system in modern software development environments cannot be overstated. The software industry is characterized by its rapid pace, with frequent updates, numerous bug reports, and a diverse workforce of developers with varying skill levels. In this dynamic landscape, it is imperative to streamline the bug resolution process, ensure efficient resource allocation, and maintain developers' morale and job satisfaction. Without automated systems, bug triaging becomes a time-consuming and error-prone task, which may lead to delays in addressing critical issues, assigning inappropriate bugs to developers, and overburdening experienced developers while underutilizing junior ones. To address these challenges, our research has culminated in creating a novel bug-triaging strategy, which combines two pivotal models. The first model, known as the BSDRM, serves as the cornerstone for automated bug triaging. BSDRM leverages machine learning algorithms and past bug reports to intelligently recommend developers for specific bug resolution tasks. Analyzing bug properties and developer expertise facilitates the efficient allocation of bugs to the most suitable developers, ensuring that critical issues are promptly addressed, and junior developers receive opportunities for skill enhancement.

In conjunction with BSDRM, we introduce DevSched, which plays a crucial role in balancing the workloads of developers. DevSched considers the workload distribution among developers, their skill levels, and the nature of the bugs. Through a systematic load-balancing algorithm, DevSched optimally allocates bugs among different categories of developers, namely Experienced Developers, New Experience Developers, and Fresh Graduates. This ensures that no developer is overburdened, critical bugs are appropriately assigned to experienced developers, and junior developers are entrusted with tasks suitable

for their level of expertise. The contributions made in this thesis are rooted in the extensive research conducted in chapters 4 and 5. This chapter serves as a platform to delineate the significant outcomes attained in this study by developing and applying BSDRM and DevSched. It elucidates how these models have effectively addressed the research question at hand. Furthermore, this chapter provides a succinct overview of the primary limitations inherent in the study. It offers insights into potential avenues for enhancing the reliability and robustness of the automatic bug-triaging model in future developments.

6.1 Summary of Results

In this section, we present a comprehensive overview of the results and findings obtained from two pivotal models: the Bug Solving Developer Recommendation Model and the Developer Scheduler. These models represent significant contributions to the realm of software development, addressing critical challenges in bug triage and developer workload management. We delve into the outcomes of each model, elucidating their respective methodologies, achievements, and implications. The findings outlined herein shed light on how these models have redefined bug assignment strategies, improved efficiency, and ensured a more equitable distribution of developer tasks, ultimately enhancing the software development process.

6.1.1 Recommend Developer Team Efficiently

Automatic bug triaging is crucial in software development to allocate resources for bug resolution efficiently and reduce software quality improvement time. Many software companies face the challenge of managing a large volume of bug reports. Often, experienced developers are overwhelmed by the sheer number of bug assignments, while newer and mid-skilled developers do not have sufficient opportunities to gain experience in bug fixing. This imbalance can lead to delays and decreased job satisfaction. Additionally, developers with varying backgrounds and experience levels, including fresh graduates and those transitioning from other domains, join the workforce, further complicating the bug assignment process. To address these issues, a systematic bug triaging system is needed to distribute bug assignments to different types of developers, allowing them to acquire knowledge and expertise in bug solving. This can create a balanced developer team consisting of experienced developers, newly experienced developers, fresh graduates, and developers from other domains. Such a diverse team is well-equipped to collaboratively tackle newly reported bugs, resulting in

efficient bug resolution and skill development for all team members. An ML-based model called the BSDRM is proposed in response to these challenges. BSDRM recommends developer teams comprising experts, medium-level fixers, and fresh graduates to handle newly reported bugs efficiently. By fostering collaboration and knowledge sharing among developers of varying experience levels and backgrounds, BSDRM aims to improve bug resolution efficiency and job satisfaction for all team members. A combined dataset comprising 56,621 developer instances is created by merging Eclipse, Mozilla, and NetBeans datasets. Their expertise does not initially categorize these developers, but three categories are introduced: ED, NED, and FG, with the manual assignment of expertise weights based on qualitative human inspection. The dataset is then split into 80% training and 20% testing sets. In the training set, two subsets are formed: Subset-1 containing dataset summaries and descriptions, and Subset-2 containing other attributes such as severity, priority, fix status, etc. Additionally, a Subset-11 is extracted from the test set, which includes dataset summaries and descriptions.

The training stage involves creating sentence-embedded models using pre-trained Bidirectional Encoder Representations from Transformers for S1 to generate a bag of developer words. Data balancing is performed on S2 to address class imbalance issues. Various classifiers (DT, ET, AdC, BC, GB, RF, KNN, NC, BNB, MNB, CoNB, GNB, SVM, SGD, LR, Pr, MLP) are trained on S2 to build a developer classifier based on experience levels. When a new bug report is received, S11 extracts a vocabulary list for developers using the pre-trained sentence embedding model. An unsupervised KNN finder is used to identify K nearest eligible developers based on the vocabulary list. Developers' experience levels are predicted using the developer classifier, and a developer team is formed with eligible individuals to address different bugs. This process ensures efficient bug assignment and resolution among developers of varying experience levels. After conducting extensive experimental analysis, it was determined that the Bagging Classifier emerged as the most robust classifier for categorizing developers based on their experience levels. BC achieved remarkable performance metrics, including the highest accuracy of 96.59%, precision of 96.62%, recall of 96.56%, F1-Score of 96.59%, a low Hamming Loss of 3.41%, a Jaccard Score of 93.42%, a high Matthews Correlation Coefficient of 94.89%, a Balanced Accuracy of 96.56%, and a Cohen's Kappa Score of 94.88%, respectively.

The Bagging Classifier has demonstrated superior performance in the bug triaging strategy due to its inherent strengths as an ensemble learning method. Ensembles, such as the Bagging Classifier, leverage the collective wisdom of multiple models to improve overall accuracy

and robustness. The technique's effectiveness is attributed to its ability to mitigate overfitting by training each base model on random subsets of the training data. Additionally, Bagging Classifier is known for its versatility, robustness to noisy data, and adaptability to different base classifiers, making it suitable for diverse datasets. In bug triaging, where the goal is to accurately classify developers based on varying experience levels, the aggregate decision-making process of Bagging Classifier may have effectively handled the intricacies of the task, resulting in outstanding accuracy of 96.59%. Additionally, Gradient Boosting and Random Forest classifiers yielded results that were very close to BC in terms of performance. Various criteria related to developer experience were thoroughly examined to evaluate the effectiveness of the Bug Solving Developer Recommendation Model. As a result, BSDRM demonstrated its ability to alleviate the heavy workload experienced by highly skilled developers, offering newly experienced and fresh graduate developer's valuable opportunities to gain deeper insights into bug-solving processes and contribute effectively to the development team.

6.1.2 Task Allocation and Load Balancing

Bug triaging ensures efficient resource allocation and timely resolution of critical issues while identifying recurring bug patterns. However, problems arise when bugs are not adequately distributed among developers. Experienced developers may become overloaded with critical bugs, while others, like mid-level developers, fresh graduates, or those from different fields, may not gain the necessary experience. This imbalance hinders promotions and reduces job satisfaction, impacting bug-fixing efficiency. To address this issue, we introduce Developer Scheduler, a task allocation model that assigns bugs to developers based on their experience, competence, and workload. DevSched utilizes Eclipse, Mozilla, and NetBeans datasets, split into three parts. It first creates developer profiles, analyzing their skills and bug-solving experience. Then, it assigns bugs by converting bug properties into vectors using TF-IDF and extracting a corpus of words from developer profiles. Bugs are allocated based on cosine similarity and predefined thresholds. Finally, load balancing is performed using a proposed algorithm, estimating developer workloads and dynamically updating them based on performance.

DevSched, applied to the Eclipse dataset, effectively balances individual developer workloads, resulting in a reduction in the standard deviation of bug assignments. The initial workload distribution exhibits some fluctuations, but the workloads become more

even after implementing the load balancing method. In the Eclipse dataset, the ED class comprises 130 developers, the NED class has 73, and the FG class includes 350 developers. Load creation initially leads to a standard deviation increase, peaking at 2.07. However, the standard deviation significantly decreases to 0.48 upon applying the load balancing method. The Mozilla dataset also benefits from DevSched, which balances developer workloads. Similar to other datasets, the initial dataset segment consists of 167 EDs, 109 NEDs, and 187 FGs, establishing thresholds for primary selection and different developer classes. Workloads are then distributed based on categorical thresholds, yielding 178 EDs, 129 NEDs, and 156 FGs. The standard deviations of these loads exhibit fluctuations, with an overall value of 6.54. However, the proposed load balancing method results in a reduced standard deviation, reaching 0.49. In the NetBeans dataset, DevSched effectively balances developer workloads using the load-balancing method. The profile creation stage involves 98 EDs, 63 NEDs, and 331 FGs. After the primary load creation based on initial thresholds, 116 EDs, 66 NEDs, and 310 FGs are assigned bugs, resulting in an overall standard deviation of 5.66 among developers. Implementing the load balancing method assigns 166 EDs, 59 NEDs, and 267 FGs to solve different bug types, reducing the standard deviation to 0.52. The Eclipse dataset exhibits the lowest standard deviation, making it well-suited for DevSched implementation. In contrast, the NetBeans dataset initially has a relatively high standard deviation, which DevSched significantly reduces with the load balancing method. The maximum standard deviation is observed in the Mozilla dataset, indicating efficient bug resolution by developers in a short time frame. The proposed DevSched enhances bug triaging by ensuring fair workload distribution among developers, including newly experienced developers, and dynamically updating their ratings based on performance.

6.2 Impact on Software Companies

The implications of the proposed method on software companies are multifaceted, influencing various aspects of their bug-triaging and development processes. This section delves into the potential impact, addressing how the newly introduced Bug Solving Developer Recommendation Model and the Developer Scheduler model can revolutionize the software development landscape.

i) Enhancing Efficiency in Developer Assignment

One of the primary impacts of these models lies in the realm of developer assignment. BSDRM leverages machine learning techniques to recommend developers for bug resolution

tasks while considering their expertise, workload, and the severity of bugs. This process leads to more efficient developer assignments, ensuring the proper developers are assigned the appropriate bugs. Consequently, software companies expect to witness a reduction in the time it takes to resolve bugs, contributing to more streamlined development pipelines and shorter release cycles.

ii) Optimizing Workload Distribution and Knowledge Sharing

DevSched, on the other hand, plays a pivotal role in optimizing the distribution of unassigned bugs among developers with varying expertise levels. By creating developer profiles, assigning bugs based on developer-corpus similarity, and dynamically updating workloads, DevSched fosters an environment of balanced workload distribution. This not only reduces the burden on experienced developers but also facilitates knowledge sharing within development teams. Fresh graduates and developers transitioning from other fields can actively participate in bug resolution, gaining valuable experience. This balanced workload distribution improves job satisfaction among developers and promotes a culture of collaboration.

iii) Accommodating Newly Joined Developers

Incorporating newly joined developers into the bug assignment process is another significant impact. Both BSDRM and DevSched ensure that newly joined developers are not overwhelmed with assignments. BSDRM classifies developers into different experience levels, ensuring that new developers are included in the recommendation process without being inundated with tasks. DevSched, by preventing the overloading of experienced developers, indirectly paves the way for new developers to engage actively in bug resolution processes. This approach promotes the integration of fresh talent into the software development team, fostering a dynamic and adaptable workforce.

iv) Enhancing Software Quality Assurance

The ultimate impact of these models is on software quality assurance. By reducing bug resolution time and ensuring that critical issues are addressed promptly, both BSDRM and DevSched contribute to enhanced software quality. In an industry inundated with bug reports from diverse software companies, these models offer a systematic approach to triaging and resolving bugs. This not only leads to higher-quality software but also enhances the reputation of software companies by delivering more reliable products to customers. Additionally, the efficiency gained through these models positively impacts the entire software development pipeline, resulting in smoother bug resolution processes.

6.3 Limitations of the Study

While the Bug Solving Developer Recommendation Model and the Developer Scheduler models offer significant advantages in bug triaging and developer assignment, they are not without limitations. It's essential to recognize these limitations to understand the constraints of their application.

Both BSDRM and DevSched heavily rely on the quality and availability of historical data. Inaccurate or incomplete data can lead to incorrect developer recommendations and suboptimal bug assignments. Ensuring high data quality and consistency is crucial for the models to perform effectively.

The performance of ML is highly sensitive to feature selection. Choosing relevant features and fine-tuning model hyperparameters is a complex and time-consuming task. The effectiveness of the models can be impacted if features are not selected or engineered appropriately.

Both models primarily focus on textual data, such as bug descriptions and developer profiles. They may not effectively handle non-textual data, such as multimedia bug reports or developer portfolios, limiting their applicability in scenarios involving diverse data types.

6.4 Future Research

As the field of software development continues to evolve, so must the methodologies and tools employed to enhance bug triaging and developer assignment processes. The Bug Solving Developer Recommendation Model and the Developer Scheduler have paved the way for more efficient and effective bug resolution practices. However, there is still ample room for growth and improvement. In this section, we delve into the future directions for these models, exploring how they can adapt to meet the ever-changing demands of the software development industry, address emerging challenges, and contribute to enhancing software quality assurance.

In the future, the Bug Solving Developer Recommendation Model will embrace a broader scope, extending its impact beyond the confines of this study. It will explore and integrate additional bug repositories, encompassing large-scale software projects, open-source initiatives, and commercial applications. By doing so, BSDRM aims to evaluate its task allocation capabilities in a more diverse and dynamic environment.

In the future, the Developer Scheduler will undergo continuous refinement and augmentation, focusing on delivering real-time services for the bug triage process. This enhancement will involve the integration of cutting-edge methodologies and technologies that enable DevSched to adapt swiftly to the ever-evolving landscape of software development.

Bibliography

1. Lee, D.G. and Seo, Y.S., 2020. Improving bug report triage performance using artificial intelligence based document generation model. Human-centric Computing and Information Sciences, 10(1), p.26.
2. Aung, T.W.W., Wan, Y., Huo, H. and Sui, Y., 2022. Multi-triage: A multi-task learning framework for bug triage. Journal of Systems and Software, 184, p.111133.
3. Kukkar, A., Kumar, Y., Sharma, A. and Sandhu, J.K., 2023. Bug Severity Classification in Software Using Ant Colony Optimization Based Feature Weighting Technique. Expert Systems with Applications, p.120573.
4. Dai, J., Li, Q., Xue, H., Luo, Z., Wang, Y. and Zhan, S., 2023. Graph collaborative filtering-based bug triaging. Journal of Systems and Software, 200, p.111667.
5. Kukkar, A., Lilhore, U.K., Frnda, J., Sandhu, J.K., Das, R.P., Goyal, N., Kumar, A., Muduli, K. and Rezac, F., 2023. ProRE: An ACO-based programmer recommendation model to precisely manage software bugs. Journal of King Saud University-Computer and Information Sciences, 35(1), pp.483-498.
6. Chauhan, R., Sharma, S. and Goyal, A., 2023. DENATURE: duplicate detection and type identification in open source bug repositories. International Journal of System Assurance Engineering and Management, pp.1-18.
7. de Oliveira Calixto, F.E., Ramalho, F., Massoni, T. and Ferreira, J.M., 2023. Investigating Bug Report Changes in Bugzilla.
8. Sarawan, K., Polpinij, J. and Luaphol, B., 2023, May. Machine Learning-Based Methods for Identifying Bug Severity Level from Bug Reports. In International Conference on Computing and Information Technology (pp. 199-208). Cham: Springer Nature Switzerland.
9. Raghuvanshi, K.K., Agarwal, A., Singh, A.K. and Jain, K., 2023. Time-dependent entropic analysis of software bugs. International Journal of System Assurance Engineering and Management, pp.1-8.

10. Dao, A.H. and Yang, C.Z., 2023. Automated Priority Prediction for Bug Reports Using Comment Intensiveness Features and SMOTE Data Balancing. International Journal of Software Engineering and Knowledge Engineering, 33(03), pp.415-433.
11. Arora, R. and Kaur, A., 2023. BugFinder: Automatic Data Extraction Approach for Bug Reports from Jira-Repositories. In Advances in Data-driven Computing and Intelligent Systems: Selected Papers from ADCIS 2022, Volume 2 (pp. 511-521). Singapore: Springer Nature Singapore.
12. Noyori, Y., Washizaki, H., Fukazawa, Y., Ooshima, K., Kanuka, H. and Nojiri, S., 2023. Deep learning and gradient-based extraction of bug report features related to bug fixing time. Frontiers in Computer Science, 5, p.1032440.
13. Peralta, S.R.O., Washizaki, H., Fukazawa, Y., Noyori, Y., Nojiri, S. and Kanuka, H., 2023, June. Analysis of Bug Report Qualities with Fixing Time using a Bayesian Network. In Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering (pp. 235-240).
14. Yan, A., Zhong, H., Song, D. and Jia, L., 2023. How do programmers fix bugs as workarounds? An empirical study on Apache projects. Empirical Software Engineering, 28(4), p.96.
15. Bansal, K., Singh, G., Malik, S. and Rohil, H., 2023. NRPredictor: an ensemble learning and feature selection based approach for predicting the non-reproducible bugs. International Journal of System Assurance Engineering and Management, 14(3), pp.989-1009.
16. Chauhan, R., Sharma, S. and Goyal, A., 2023. DENATURE: duplicate detection and type identification in open source bug repositories. International Journal of System Assurance Engineering and Management, pp.1-18.
17. Liang, H. and Wei, Q., 2023, August. A hybrid approach for developer recommendation based on social network. In Second International Conference on Electronic Information Technology (EIT 2023) (Vol. 12719, pp. 759-766). SPIE.
18. Tabassum, N., Namoun, A., Alyas, T., Tufail, A., Taqi, M. and Kim, K.H., 2023. Classification of Bugs in Cloud Computing Applications Using Machine Learning Techniques. Applied Sciences, 13(5), p.2880.
19. Mukherjee, U. and Rahman, M.M., 2023. Answering Follow-up Questions on Bug Reports with Structured Information Retrieval and Deep Learning. arXiv preprint arXiv:2304.12494.

20. Xu, Y., Liu, C., Li, Y., Xie, Q. and Choi, H.D., 2023, March. A Method of Component Prediction for Crash Bug Reports Using Component-Based Features and Machine Learning. In 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) (pp. 773-777). IEEE.
21. Qian, C., Zhang, M., Nie, Y., Lu, S. and Cao, H., 2023. A Survey on Bug Deduplication and Triage Methods from Multiple Points of View. Applied Sciences, 13(15), p.8788.
22. Gomes, L., da Silva Torres, R. and Côrtes, M.L., 2023. BERT-and TF-IDF-based feature extraction for long-lived bug prediction in FLOSS: a comparative study. Information and Software Technology, 160, p.107217.
23. Life Cycles of Bug. [Online]. Available From: https://www.ques10.com/p/48607/life-cycle-of-bugs-1/ [retrieved 10 September, 2023].
24. Software Testing. [Online]. Available From: https://www.softwaretestinghelp.com/bug-life-cycle/ [retrieved 08 September, 2023]
25. Samir, M., Sherief, N. and Abdelmoez, W., 2023. Improving Bug Assignment and Developer Allocation in Software Engineering through Interpretable Machine Learning Models. Computers, 12(7), p.128.
26. Rao, N.R. and Suresh, K., 2023. Enhanced Bug Localization through Version Tag Embedding: A Comprehensive Approach to Efficient Software Development. International Journal of Intelligent Systems and Applications in Engineering, 11(6s), pp.417-427.
27. Jonsson, L., Borg, M., Broman, D., Sandahl, K., Eldh, S. and Runeson, P., 2016. Automated bug assignment: Ensemble-based machine learning in large scale industrial contexts. Empirical Software Engineering, 21, pp.1533-1578.
28. 28. Eclipse. [Online]. Available From: https://wiki.eclipse.org/Mylyn/User_Guide [retrieved 18 November, 2021]
29. Gupta, M. and Sureka, A., 2014, February. Nirikshan: Mining bug report history for discovering process maps, inefficiencies and inconsistencies. In Proceedings of the 7th India Software Engineering Conference (pp. 1-10).
30. Rajalakshmi, R., Selvaraj, S. and Vasudevan, P., 2023. Hottest: Hate and offensive content identification in Tamil using transformers and enhanced stemming. Computer Speech & Language, 78, p.101464.
31. Alshammari, N.O. and Alharbi, F.D., 2022. Combining a Novel Scoring Approach with Arabic Stemming Techniques for Arabic Chatbots Conversation Engine. Transactions on Asian and Low-Resource Language Information Processing, 21(4), pp.1-21.

32. Karaa, W.B.A. and Gribâa, N., 2013. Information retrieval with porter stemmer: a new version for English. In Advances in Computational Science, Engineering and Information Technology: Proceedings of the Third International Conference on Computational Science, Engineering and Information Technology (CCSEIT-2013), KTO Karatay University, June 7-9, 2013, Konya, Turkey-Volume 1 (pp. 243-254). Springer International Publishing.
33. Porter, M.F., 2001. Snowball: A language for stemming algorithms.
34. Jodha, R. and Dadheech, A., 2019. Analysis and Evaluation of Unstructured data based on Stemming Algorithms. American International Journal of Research in Formal, Applied & Natural Sciences AIJRFANS, pp.19-201.
35. Karaa, W.B.A., 2013. A new stemmer to improve information retrieval. International Journal of Network Security & Its Applications, 5(4), p.143.
36. Jumadi, J., Maylawati, D.S., Pratiwi, L.D. and Ramdhani, M.A., 2021, March. Comparison of Nazief-Adriani and Paice-Husk algorithm for Indonesian text stemming process. In IOP Conference Series: Materials Science and Engineering (Vol. 1098, No. 3, p. 032044). IOP Publishing.
37. Jalil, M.M., Ismailov, A., Abd Rahim, N.H. and Abdullah, Z., 2017. The Development of the Uzbek Stemming Algorithm. Advanced Science Letters, 23(5), pp.4171-4174.
38. Faheem, M.R., Anees, T. and Hussain, M., 2022. Keywords and Spatial Based Indexing for Searching the Things on Web. KSII Transactions on Internet & Information Systems, 16(5).
39. Kumar, H., Mahindru, R. and Kar, D., 2022, November. Metadata-based retrieval for resolution recommendation in AIOps. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (pp. 1379-1389).
40. Cabasag, C.J., Fagan, P.J., Ferlay, J., Vignat, J., Laversanne, M., Liu, L., van der Aa, M.A., Bray, F. and Soerjomataram, I., 2022. Ovarian cancer today and tomorrow: A global assessment by world region and Human Development Index using GLOBOCAN 2020. International Journal of Cancer, 151(9), pp.1535-1541.
41. Pu, W., Raman, A.A.A., Hamid, M.D., Gao, X. and Buthiyappan, A., 2023. Inherent safety concept based proactive risk reduction strategies: A review. Journal of Loss Prevention in the Process Industries, p.105133.

42. Liu, W., Gan, Z., Xi, T., Du, Y., Wu, J., He, Y., Jiang, P., Liu, X. and Lai, X., 2023. A semantic and intelligent focused crawler based on semantic vector space model and membrane computing optimization algorithm. Applied Intelligence, 53(7), pp.7390-7407.
43. Widaningrum, I., Mustikasari, D., Arifin, R., Tsaqila, S.L. and Fatmawati, D., 2022. Algoritma Term Frequency–Inverse Document Frequency (TF-IDF) dan K-Means Clustering Untuk Menentukan Kategori Dokumen. Prosiding SISFOTEK, 6(1), pp.145-149.
44. Yu, T., Liu, J., Yang, Y., Li, Y., Fei, H. and Li, P., 2022, August. EGM: enhanced graph-based model for large-scale video advertisement search. In Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (pp. 4443-4451).
45. Wang, R., Li, J. and Bai, R., 2023. Prediction and Analysis of Container Terminal Logistics Arrival Time Based on Simulation Interactive Modeling: A Case Study of Ningbo Port. Mathematics, 11(15), p.3271.
46. Hu, H., Zhang, H., Xuan, J. and Sun, W., 2014, November. Effective bug triage based on historical bug-fix information. In 2014 IEEE 25th international symposium on software reliability engineering (pp. 122-132). IEEE.
47. Xie, X., Zhang, W., Yang, Y. and Wang, Q., 2012, September. Dretom: Developer recommendation based on topic models for bug resolution. In Proceedings of the 8th international conference on predictive models in software engineering (pp. 19-28).
48. Zhang, W., Han, G. and Wang, Q., 2014, November. Butter: An approach to bug triage with topic modeling and heterogeneous network analysis. In 2014 International Conference on Cloud Computing and Big Data (pp. 62-69). IEEE.
49. Zhang, T., Yang, G., Lee, B. and Lua, E.K., 2014, December. A novel developer ranking algorithm for automatic bug triage using topic model and developer relations. In 2014 21st Asia-Pacific Software Engineering Conference (Vol. 1, pp. 223-230). IEEE.
50. Zhang, W., Wang, S. and Wang, Q., 2016. BAHA: A novel approach to automatic bug report assignment with topic modeling and heterogeneous network analysis. Chinese Journal of Electronics, 25(6), pp.1011-1018.
51. Kagdi, H., Gethers, M., Poshyvanyk, D. and Hammad, M., 2012. Assigning change requests to software developers. Journal of software: Evolution and Process, 24(1), pp.3-33.
52. Shokripour, R., Kasirun, Z.M., Zamani, S. and Anvik, J., 2012, November. Automatic bug assignment using information extraction methods. In 2012 International conference

on advanced computer science applications and technologies (ACSAT) (pp. 144-149). IEEE.

53. Shokripour, R., Anvik, J., Kasirun, Z.M. and Zamani, S., 2013, May. Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation. In 2013 10th working conference on mining software repositories (MSR) (pp. 2-11). IEEE.
54. Linares-Vásquez, M., Hossen, K., Dang, H., Kagdi, H., Gethers, M. and Poshyvanyk, D., 2012, September. Triaging incoming change requests: Bug or commit history, or code authorship?. In 2012 28th IEEE International Conference on Software Maintenance (ICSM) (pp. 451-460). IEEE.
55. Naguib, H., Narayan, N., Brügge, B. and Helal, D., 2013, May. Bug report assignee recommendation using activity profiles. In 2013 10th Working Conference on Mining Software Repositories (MSR) (pp. 22-30). IEEE.
56. Yang, G., Zhang, T. and Lee, B., 2014, July. Towards semi-automatic bug triage and severity prediction based on topic model and multi-feature of bug reports. In 2014 IEEE 38th Annual Computer Software and Applications Conference (pp. 97-106). IEEE.
57. Wang, S., Zhang, W. and Wang, Q., 2014, September. FixerCache: Unsupervised caching active developers for diverse bug triage. In Proceedings of the 8th ACM/IEEE international symposium on empirical software engineering and measurement (pp. 1-10).
58. Xia, X., Lo, D., Ding, Y., Al-Kofahi, J.M., Nguyen, T.N. and Wang, X., 2016. Improving automated bug triaging with specialized topic model. IEEE Transactions on Software Engineering, 43(3), pp.272-297.
59. Lee, D.G. and Seo, Y.S., 2020. Improving bug report triage performance using artificial intelligence based document generation model. Human-centric Computing and Information Sciences, 10(1), p.26.
60. Zhang, W., Cui, Y. and Yoshida, T., 2017. En-lda: An novel approach to automatic bug report assignment with entropy optimized latent dirichlet allocation. Entropy, 19(5), p.173.
61. Yadav, A., Singh, S.K. and Suri, J.S., 2019. Ranking of software developers based on expertise score for bug triaging. Information and Software Technology, 112, pp.1-17.
62. Banitaan, S. and Alenezi, M., 2013, December. Decoba: Utilizing developers communities in bug assignment. In 2013 12th International Conference on Machine Learning and Applications (Vol. 2, pp. 66-71). IEEE.

63. Zhang, W., Wang, S., Yang, Y. and Wang, Q., 2013, November. Heterogeneous network analysis of developer contribution in bug repositories. In 2013 International Conference on Cloud and Service Computing (pp. 98-105). IEEE.
64. Zhang, T. and Lee, B., 2013, March. A hybrid bug triage algorithm for developer recommendation. In Proceedings of the 28th annual ACM symposium on applied computing (pp. 1088-1094).
65. Zhang, T. and Lee, B., 2012. An automated bug triage approach: A concept profile and social network based developer recommendation. In Intelligent Computing Technology: 8th International Conference, ICIC 2012, Huangshan, China, July 25-29, 2012. Proceedings 8 (pp. 505-512). Springer Berlin Heidelberg.
66. Kumari, M., Misra, A., Misra, S., Fernandez Sanz, L., Damasevicius, R. and Singh, V.B., 2019. Quantitative quality evaluation of software products by considering summary and comments entropy of a reported bug. Entropy, 21(1), p.91.
67. Etemadi, V., Bushehrian, O., Akbari, R. and Robles, G., 2021. A scheduling-driven approach to efficiently assign bug fixing tasks to developers. Journal of Systems and Software, 178, p.110967.
68. Almhana, R. and Kessentini, M., 2021. Considering dependencies between bug reports to improve bugs triage. Automated Software Engineering, 28, pp.1-26.
69. Jahanshahi, H., Chhabra, K., Cevik, M. and Başar, A., 2021. DABT: A dependency-aware bug triaging method. In Evaluation and Assessment in Software Engineering (pp. 221-230).
70. Bhattacharya, P. and Neamtiu, I., 2010, September. Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging. In 2010 IEEE International Conference on Software Maintenance (pp. 1-10). IEEE.
71. Tamrawi, A., Nguyen, T.T., Al-Kofahi, J. and Nguyen, T.N., 2011, May. Fuzzy set-based automatic bug triaging (NIER track). In Proceedings of the 33rd international conference on software engineering (pp. 884-887).
72. Anvik, J. and Murphy, G.C., 2011. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. ACM Transactions on Software Engineering and Methodology (TOSEM), 20(3), pp.1-35.
73. Xuan, J., Jiang, H., Ren, Z. and Zou, W., 2012, June. Developer prioritization in bug repositories. In 2012 34th International Conference on Software Engineering (ICSE) (pp. 25-35). IEEE.

74. Banitaan, S. and Alenezi, M., 2013, June. Tram: An approach for assigning bug reports using their metadata. In 2013 Third International Conference on Communications and Information Technology (ICCIT) (pp. 215-219). IEEE.
75. Alenezi, M., Magel, K. and Banitaan, S., 2013. Efficient Bug Triaging Using Text Mining. J. Softw., 8(9), pp.2185-2190.
76. Xuan, J., Jiang, H., Hu, Y., Ren, Z., Zou, W., Luo, Z. and Wu, X., 2014. Towards effective bug triage with software data reduction techniques. IEEE transactions on knowledge and data engineering, 27(1), pp.264-280.
77. Jonsson, L., Borg, M., Broman, D., Sandahl, K., Eldh, S. and Runeson, P., 2016. Automated bug assignment: Ensemble-based machine learning in large scale industrial contexts. Empirical Software Engineering, 21, pp.1533-1578.
78. Florea, A.C., Anvik, J. and Andonie, R., 2017. Spark-based cluster implementation of a bug report assignment recommender system. In Artificial Intelligence and Soft Computing: 16th International Conference, ICAISC 2017, Zakopane, Poland, June 11-15, 2017, Proceedings, Part II 16 (pp. 31-42). Springer International Publishing.
79. Alenezi, M., Banitaan, S. and Zarour, M., 2018. Using categorical features in mining bug tracking systems to assign bug reports. arXiv preprint arXiv:1804.07803.
80. Sarkar, A., Rigby, P.C. and Bartalos, B., 2019, September. Improving bug triaging with high confidence predictions at ericsson. In 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME) (pp. 81-91). IEEE.
81. Anvik, J., Hiew, L. and Murphy, G.C., 2006, May. Who should fix this bug?. In Proceedings of the 28th international conference on Software engineering (pp. 361-370).
82. Peng, X., Zhou, P., Liu, J. and Chen, X., 2017, July. Improving Bug Triage with Relevant Search. In SEKE (pp. 123-128).
83. Nagwani, N.K. and Verma, S., 2012, January. Predicting expert developers for newly reported bugs using frequent terms similarities of bug attributes. In 2011 Ninth International Conference on ICT and Knowledge Engineering (pp. 113-117). IEEE.
84. Lee, S.R., Heo, M.J., Lee, C.G., Kim, M. and Jeong, G., 2017, August. Applying deep learning based automatic bug triager to industrial projects. In Proceedings of the 2017 11th Joint Meeting on foundations of software engineering (pp. 926-931).
85. Choquette-Choo, C.A., Sheldon, D., Proppe, J., Alphonso-Gibbs, J. and Gupta, H., 2019, December. A multi-label, dual-output deep neural network for automated bug triaging. In 2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA) (pp. 937-944). IEEE.

86. Mani, S., Sankaran, A. and Aralikatte, R., 2019, January. Deeptriage: Exploring the effectiveness of deep learning for bug triaging. In Proceedings of the ACM India joint international conference on data science and management of data (pp. 171-179).
87. Guo, S., Zhang, X., Yang, X., Chen, R., Guo, C., Li, H. and Li, T., 2020. Developer activity motivated bug triaging: via convolutional neural network. Neural Processing Letters, 51, pp.2589-2606.
88. Zaidi, S.F.A., Awan, F.M., Lee, M., Woo, H. and Lee, C.G., 2020. Applying convolutional neural networks with different word representation techniques to recommend bug fixers. IEEE Access, 8, pp.213729-213747.
89. Zaidi, S.F.A. and Lee, C.G., 2021, January. Learning graph representation of bug reports to triage bugs using graph convolution network. In 2021 International Conference on Information Networking (ICOIN) (pp. 504-507). IEEE.
90. Choquette-Choo, C.A., Sheldon, D., Proppe, J., Alphonso-Gibbs, J. and Gupta, H., 2019, December. A multi-label, dual-output deep neural network for automated bug triaging. In 2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA) (pp. 937-944). IEEE.
91. Jeong, G., Kim, S. and Zimmermann, T., 2009, August. Improving bug triage with bug tossing graphs. In Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (pp. 111-120).
92. Chen, L., Wang, X. and Liu, C., 2011. An Approach to Improving Bug Assignment with Bug Tossing Graphs and Bug Similarities. J. Softw., 6(3), pp.421-427.
93. Bhattacharya, P. and Neamtiu, I., 2010, September. Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging. In 2010 IEEE International Conference on Software Maintenance (pp. 1-10). IEEE.
94. Baysal, O., Godfrey, M.W. and Cohen, R., 2009, May. A bug you like: A framework for automated assignment of bugs. In 2009 IEEE 17th International Conference on Program Comprehension (pp. 297-298). IEEE.
95. Hossen, M.K., 2013. Triaging incoming change requests: bug or commit history, or code authorship?
96. Aggarwal, K., Timbers, F., Rutgers, T., Hindle, A., Stroulia, E. and Greiner, R., 2017. Detecting duplicate bug reports with software engineering domain knowledge. Journal of Software: Evolution and Process, 29(3), p.e1821.

97. Matter, D., Kuhn, A. and Nierstrasz, O., 2009, May. Assigning bug reports using a vocabulary-based expertise model of developers. In 2009 6th IEEE international working conference on mining software repositories (pp. 131-140). IEEE.
98. Shokripour, R., Anvik, J., Kasirun, Z.M. and Zamani, S., 2015. A time-based approach to automatic bug report assignment. Journal of Systems and Software, 102, pp.109-122.
99. Park, J.W., Lee, M.W., Kim, J., Hwang, S.W. and Kim, S., 2016. Cost-aware triage ranking algorithms for bug reporting systems. Knowledge and Information Systems, 48, pp.679-705.
100. Dedík, V. and Rossi, B., 2016, August. Automated bug triaging in an industrial context. In 2016 42th Euromicro conference on software engineering and advanced applications (SEAA) (pp. 363-367). IEEE.
101. Zhang, W., Wang, S. and Wang, Q., 2016. KSAP: An approach to bug report assignment using KNN search and heterogeneous proximity. Information and software technology, 70, pp.68-84.
102. Pedregosa, F., et al.: Scikit-learn: Machine learning in python. J. Mach. Learn. Res. 12, 2825–2830 (2011)
103. Prottasha, N.J., et al.: Transfer learning for sentiment analysis using bert based supervised fine-tuning. Sensors 22(11), 4157 (2022)
104. Satu, M.S., et al.: Tclustvid: a novel machine learning classification model to investigate topics and sentiment in covid-19 tweets. Knowledge-Based Systems, p. 107126 (2021)
105. Wolf, T., et al.: Huggingface's transformers: State-of-the-art natural language processing. arXiv preprint arXiv:1910.03771 (2019)
106. Podgorelec, V., Kokol, P., Stiglic, B. and Rozman, I., 2002. Decision trees: an overview and their use in medicine. *Journal of medical systems*, *26*, pp.445-463.
107. Costa, V.G. and Pedreira, C.E., 2023. Recent advances in decision trees: An updated survey. *Artificial Intelligence Review*, *56*(5), pp.4765-4800.
108. Blockeel, H., Devos, L., Frénay, B., Nanfack, G. and Nijssen, S., 2023. Decision trees: from efficient prediction to responsible AI. *Frontiers in Artificial Intelligence*, *6*.
109. El Bilali, A., Abdeslam, T., Ayoub, N., Lamane, H., Ezzaouini, M.A. and Elbeltagi, A., 2023. An interpretable machine learning approach based on DNN, SVR, Extra Tree, and XGBoost models for predicting daily pan evaporation. *Journal of Environmental Management*, *327*, p.116890.

110. Ramakrishna, M.T., Venkatesan, V.K., Izonin, I., Havryliuk, M. and Bhat, C.R., 2023. Homogeneous Adaboost Ensemble Machine Learning Algorithms with Reduced Entropy on Balanced Data. *Entropy*, *25*(2), p.245.

111. Lan, Y., Zhang, Y. and Lin, W., 2023. Diagnosis algorithms for indirect bridge health monitoring via an optimized AdaBoost-linear SVM. *Engineering Structures*, *275*, p.115239.

112. Hao, L. and Huang, G., 2023. An improved AdaBoost algorithm for identification of lung cancer based on electronic nose. *Heliyon*, *9*(3).

113. Sharma, D. and Selwal, A., 2023. SFincBuster: Spoofed fingerprint buster via incremental learning using leverage bagging classifier. *Image and Vision Computing*, *135*, p.104713.

114. Chandramouli, A., Hyma, V.R., Tanmayi, P.S., Santoshi, T.G. and Priyanka, B., 2023. Diabetes prediction using Hybrid Bagging Classifier. *Entertainment Computing*, *47*, p.100593.

115. Janapareddy, D. and Yenduri, N.C., 2023. Credit Card Approval Prediction: A comparative analysis between logistic regressionclassifier, random forest classifier, support vectorclassifier with ensemble bagging classifier.

116. Bentéjac, C., Csörgő, A. and Martínez-Muñoz, G., 2021. A comparative analysis of gradient boosting algorithms. *Artificial Intelligence Review*, *54*, pp.1937-1967.

117. Dorogush, A.V., Ershov, V. and Gulin, A., 2018. CatBoost: gradient boosting with categorical features support. *arXiv preprint arXiv:1810.11363*.

118. Khorshid, S.F. and Abdulazeez, A.M., 2021. Breast cancer diagnosis based on k-nearest neighbors: a review. *PalArch's Journal of Archaeology of Egypt/Egyptology*, *18*(4), pp.1927-1951.

119. Boateng, E.Y., Otoo, J. and Abaye, D.A., 2020. Basic tenets of classification algorithms K-nearest-neighbor, support vector machine, random forest and neural network: a review. *Journal of Data Analysis and Information Processing*, *8*(4), pp.341-357.

120. Kumbure, M.M., Luukka, P. and Collan, M., 2020. A new fuzzy k-nearest neighbor classifier based on the Bonferroni mean. *Pattern Recognition Letters*, *140*, pp.172-178.

121. France, M.T., Ma, B., Gajer, P., Brown, S., Humphrys, M.S., Holm, J.B., Waetjen, L.E., Brotman, R.M. and Ravel, J., 2020. VALENCIA: a nearest centroid classification method for vaginal microbial communities based on composition. *Microbiome*, *8*, pp.1-15.

122. Johri, S., Debnath, S., Mocherla, A., Singk, A., Prakash, A., Kim, J. and Kerenidis, I., 2021. Nearest centroid classification on a trapped ion quantum computer. *npj Quantum Information*, *7*(1), p.122.

123. Artur, M., 2021. Review the performance of the Bernoulli Naïve Bayes Classifier in Intrusion Detection Systems using Recursive Feature Elimination with Cross-validated selection of the best number of features. *Procedia computer science*, *190*, pp.564-570.

124. Singh, M., Bhatt, M.W., Bedi, H.S. and Mishra, U., 2020. WITHDRAWN: Performance of bernoulli's naive bayes classifier in the detection of fake news.

125. Hossain, E., Sharif, O. and Moshiul Hoque, M., 2021. Sentiment polarity detection on bengali book reviews using multinomial naive bayes. In *Progress in Advanced Computing and Intelligent Engineering: Proceedings of ICACIE 2020* (pp. 281-292). Singapore: Springer Singapore.

126. Pham, B.T., Phong, T.V., Nguyen, H.D., Qi, C., Al-Ansari, N., Amini, A., Ho, L.S., Tuyen, T.T., Yen, H.P.H., Ly, H.B. and Prakash, I., 2020. A comparative study of kernel logistic regression, radial basis function classifier, multinomial naïve bayes, and logistic model tree for flash flood susceptibility mapping. *Water*, *12*(1), p.239.

127. Farisi, A.A., Sibaroni, Y. and Al Faraby, S., 2019, March. Sentiment analysis on hotel reviews using Multinomial Naïve Bayes classifier. In *Journal of Physics: Conference Series* (Vol. 1192, No. 1, p. 012024). IOP Publishing.

128. Anagaw, A. and Chang, Y.L., 2019. A new complement naïve Bayesian approach for biomedical data classification. *Journal of Ambient Intelligence and Humanized Computing*, *10*, pp.3889-3897.

129. Ali, L., Khan, S.U., Golilarz, N.A., Yakubu, I., Qasim, I., Noor, A. and Nour, R., 2019. A feature-driven decision support system for heart failure prediction based on statistical model and Gaussian naive bayes. *Computational and Mathematical Methods in Medicine*, *2019*.

130. Islam, R., Devnath, M.K., Samad, M.D. and Al Kadry, S.M.J., 2022. GGNB: Graph-based Gaussian naive Bayes intrusion detection system for CAN bus. *Vehicular Communications*, *33*, p.100442.

131. Cataldi, L., Tiberi, L. and Costa, G., 2021. Estimation of MCS intensity for Italy from high quality accelerometric data, using GMICEs and Gaussian Naïve Bayes Classifiers. *Bulletin of Earthquake Engineering*, *19*, pp.2325-2342.

132. Srimaneekarn, N., Hayter, A., Liu, W. and Tantipoj, C., 2022. Binary response analysis using logistic regression in dentistry. *International Journal of Dentistry*, *2022*.

133. Song, X., Liu, X., Liu, F. and Wang, C., 2021. Comparison of machine learning and logistic regression models in predicting acute kidney injury: A systematic review and meta-analysis. *International journal of medical informatics*, *151*, p.104484.

134. Du, K.L., Leung, C.S., Mow, W.H. and Swamy, M.N.S., 2022. Perceptron: Learning, generalization, model selection, fault tolerance, and role in the deep learning era. *Mathematics*, *10*(24), p.4730.

135. Desai, M. and Shah, M., 2021. An anatomization on breast cancer detection and diagnosis employing multi-layer perceptron neural network (MLP) and Convolutional neural network (CNN). *Clinical eHealth*, *4*, pp.1-11.

136. Heidari, A.A., Faris, H., Mirjalili, S., Aljarah, I. and Mafarja, M., 2020. Ant lion optimizer: theory, literature review, and application in multi-layer perceptron neural networks. *Nature-Inspired Optimizers: Theories, Literature Reviews and Applications*, pp.23-46.

137. Khatun, A., Sakib, K.: A bug assignment approach combining expertise and recency of both bug fixing and source commits. In: ENASE, pp. 351–358 (2018)

138. Khatun, A.: A team allocation technique ensuring bug assignment to existing and new developers using their recency and expertise (2017)