Building an intelligent system for predicting and fixing performance defects

by

Guoliang Zhao

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Doctor of Philosophy

Queen's University
Kingston, Ontario, Canada
August 2022

Abstract

Software systems have been playing an essential role in supporting our daily activities. Performance anomalies are unexpected performance degradation that deviates from the normal behaviors of software systems. Performance anomalies can cause a dramatically negative impact on users' satisfaction. The increasing scale and complexity of software systems make these systems prone to performance anomalies that are caused by various reasons, such as, misconfiguration, hardware failures, resource contentions, and performance defects. To help development and operation teams to maintain the performance of software systems, prior studies propose various approaches to detect performance anomalies and performance defects. However, prior detection approaches cannot predict the performance anomalies ahead of time; such limitation causes an inevitable delay in taking corrective actions to prevent performance anomalies from happening.

To help developers and operators to prevent anomalies from happening, in this thesis, we conduct a set of studies to predict performance anomalies from run-time monitoring data and predict performance defects at the development phase. More specifically, our approach consists of four aspects: (1) we propose an approach to predict performance anomalies in software systems by analyzing run-time monitoring data; (2) we propose an approach that can predict a large variety of performance

defects during development phase; (3) we provide a generic approach that predicts methods with any types of defects (e.g., performance and non-performance defects) and their fixing effort; and (4) we propose an approach to prioritize pull requests to help reviewers review code changes. Through a series of experiments, we observe that our approaches can help the development and operation teams to avoid performance anomalies at the run-time, and capture and fix performance defects at the development phase.

Related Publications

The publications related to the early versions of this thesis are listed below:

- **Predicting Performance Anomalies in Software Systems at Run-time (Chapter 3)**. Guoliang Zhao, Safwat Hassan, Ying Zou, Derek Truong, and Toby Corbin. In ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 30, no. 3, pp. 1-33, 2021.
- **Improving the pull requests review process using learning-to-rank algorithms (Chapter 6)**. Guoliang Zhao, Daniel Alencar da Costa, and Ying Zou. In Empirical Software Engineering (EMSE), vol. 24, no. 4, pp. 2140-2170, 2019.

I am the primary author of all the above publications. The research papers are co-authored with Dr. Ying Zou, Dr. Daniel Alencar da Costa, Dr. Safwat Hassan, Derek Truong, and Toby Corbin. Dr. Ying Zou supervised all of the related research. The co-authors participated in meetings and provided me with feedback and suggestions to improve my work.

Acknowledgments

Three years ago, when my academic brother, Dr. Yu Zhao, finished his PhD defense, he told me that "soon this day will come to you" in the same way as his academic brother, Dr. Feng Zhang, had told him. Finally, this day comes to me and I would like to express my appreciation to the many people that helped me on my master and PhD journey.

First and foremost, I would like to sincerely thank my supervisor, Dr. Ying Zou. Back to five years ago, she saw the potentials in me and offered me the chance to do my graduate studies at Queen's University, Canada. I would never become the researcher that I am today without her constant mentorship, guidance, and support in the past five years. In the environment provided by Dr. Ying Zou, I learned not only the technical skills and the systematic methods in conducting research, but more importantly, the communication skills to build social connections and the mindset about pushing myself to seek opportunities to improve.

I am grateful and lucky to work with the great group of researchers at the Software Evolution & Analytics Lab (SEAL), including Dr. Daniel Alencar da Costa, Dr. Safwat Hassan, Dr. Stefanos Georgiou, Dr. Ehsan Noei, Dr. Yu Zhao, Dr. Mariam El Mezouar, Osama Ehsan, Shayan Noei, Omar El Zarif, Maram Assi, Jiawen Liu, Muhammad Raza, Aidan Yang, Jieyeon Woo, Tanghaoran Zhang, Chunli Yu, Zitong

Su, Bihui Jin, Yiping Jia, and Fangjian Lei. Dr. Daniel Alencar da Costa was the first Post Doc in my graduate studies. He showed me the way of conducting research, working out, and many other aspects in life. His passion about research and life has been a big motivation for me to pursue my life goals. I also want to thank my best friends for the past five years: Ai Li, Mohammad Salloum, Hayden Banting, Wang Xu, Chlo Xu, Gongyu Huang, and Dmitri Liu. Thanks for the support that you gave me in the last five years. Special thanks go to Chlo Xu, who encouraged me and helped me in my most depressing time. I would also like to thank Ai Li, who has been close as a family member to me since the first time we met five years ago. Meeting you when I just moved to Canada is forever one of my best life experiences.

I would like to express my deepest gratitude to my previous master's supervisor, Qiongsheng Zhang, in University of Petroleum (Huadong), China. I was extremely fortunate to meet her in my third year undergraduate study. The support and help that I have received from her ever since then were beyond words. She played great roles in my development as a researcher and as a person.

I would like to thank my parents whom without their support and love this thesis would have not been possible. I dedicate this thesis to my beloved parents.

Statement of Originality

With this statement I, Guoliang Zhao, hereby declare that the described research in this thesis is my own work. Any publications, ideas, and inventions from others have been properly referenced.

Contents

Abstract i

Related Publications iii

Acknowledgments iv

Statement of Originality vi

Contents vii

List of Tables xi

List of Figures xv

Chapter 1: Introduction 1
- 1.1 Background . . . 1
  - 1.1.1 Performance Anomalies . . . 1
  - 1.1.2 Performance Defects . . . 2
  - 1.1.3 Software Performance Monitoring Process . . . 3
  - 1.1.4 Pull-based Development . . . 4
  - 1.1.5 Overview of Detecting and Fixing Performance Anomalies . . . 6
- 1.2 Research Challenges . . . 7
- 1.3 Thesis Statement . . . 9
- 1.4 Thesis Objectives . . . 10
- 1.5 Thesis Overview . . . 13

Chapter 2: Related Work 15
- 2.1 Performance Anomaly Detection . . . 15
- 2.2 Performance Anomaly Prediction . . . 20
- 2.3 Performance Defect Prediction . . . 22
- 2.4 Predicting Fixing Effort of Defects . . . 28
- 2.5 Improving Pull Request Review Process . . . 31

Chapter 3: Predicting Performance Anomalies in Software Systems by Analyzing Run-time Monitoring Data **35**
3.1 Introduction . . . 36
3.2 System Design . . . 42
3.2.1 Monitoring Applications . . . 42
3.2.2 Architecture of our approach . . . 44
3.2.3 Training our Predictive LSTM Neural Network . . . 46
3.2.4 Performance Anomaly Prediction Using the Trained LSTM Neural Network . . . 48
3.3 Data Collection . . . 49
3.3.1 Data Collection for Elasticsearch . . . 54
3.3.2 Data Collection for Hadoop . . . 57
3.4 Evaluation . . . 57
3.4.1 RQ1. What is the performance of our approach for predicting performance anomalies? . . . 58
3.4.2 RQ2. How early our approach can raise a performance anomaly warning before the anomaly occurs? . . . 70
3.4.3 RQ3. Could our approach be used to predict anomalies happened in the real world? . . . 74
3.5 Discussion . . . 82
3.5.1 Implementing our approach to monitor applications in the industrial environment . . . 82
3.5.2 The usage scenarios of our approach . . . 83
3.5.3 The limitations of our approach . . . 85
3.6 Threats to Validity . . . 86
3.7 Summary . . . 87

Chapter 4: Enhancing Performance Defect Prediction Using Performance Code Metrics **89**
4.1 Introduction . . . 90
4.2 Preliminary Study. What are the characteristics of performance defects in comparison to non-performance defects? . . . 93
4.3 Experiment Setup . . . 100
4.3.1 Selecting projects . . . 101
4.3.2 Process of manual analyses . . . 103
4.3.3 Identifying performance defect fixing commits . . . 104
4.3.4 Identifying the files with performance defects . . . 104
4.3.5 Capturing the code characteristics of performance defects . . . 106
4.3.6 Calculating the performance code metrics . . . 112
4.3.7 Calculating the code and process metrics . . . 117

4.4 Experiment Results . . . 119
4.4.1 RQ1. What is the performance of our approach in predicting performance defects at file level? . . . 119
4.4.2 RQ2. Which group of metrics affect the performance of our models the most? . . . 127
4.4.3 RQ3. What are the different types of performance defects that our approach can predict? . . . 132
4.5 Discussion . . . 135
4.5.1 Challenges in fixing performance defects . . . 135
4.5.2 Limitations of our approach . . . 138
4.5.3 Usage of our approach . . . 142
4.5.4 Predicting performance bugs at a fine grained level . . . 143
4.5.5 Conducting cross-project performance bugs prediction . . . 143
4.6 Threats to Validity . . . 144
4.7 Summary . . . 146

Chapter 5: An Empirical Study On Predicting Buggy Methods And Their Fixing Effort 148
5.1 Introduction . . . 149
5.2 Experiment Setup . . . 153
5.2.1 Projects selection . . . 153
5.2.2 Collecting methods from the recent releases of projects . . . 155
5.2.3 Applying CodeBERT to predict buggy methods . . . 158
5.2.4 Collecting historical buggy methods and label fixing effort categories . . . 161
5.3 Results and Analysis . . . 163
5.3.1 RQ1. What is the performance of our approach for predicting the fixing effort categories of buggy methods? . . . 164
5.3.2 RQ2. What are the most significant groups of metrics that affect the prediction performance of our models? . . . 171
5.4 Threats to Validity . . . 175
5.5 Summary . . . 177

Chapter 6: Improving the Pull Requests Review Process Using Learning-to-rank Algorithms 178
6.1 Introduction . . . 179
6.2 Experiment Setup . . . 183
6.2.1 Collecting data . . . 184
6.2.2 Metrics Calculation . . . 186
6.2.3 Correlation and redundancy analysis . . . 190

6.2.4 PRs labeling . . . 192
6.3 Results . . . 193
6.3.1 RQ1. Which learning-to-rank algorithm is the most suitable for ranking PRs? . . . 193
6.3.2 RQ2. Is our approach effective to rank PRs that can receive decisions quickly? . . . 200
6.3.3 RQ3. What are the most significant metrics that affect the ranking? . . . 204
6.4 Discussion . . . 207
6.4.1 Implication of our finding . . . 207
6.4.2 Threshold to remove noise PRs . . . 208
6.4.3 Types of the recommended PRs . . . 209
6.4.4 PRs reviewed by reviewers of different experience levels . . . 211
6.5 Threats to validity . . . 214
6.6 Summary . . . 216

Chapter 7: Conclusions and Future Work **217**
7.1 Contributions . . . 217
7.2 Future Work . . . 221

Bibliography **225**

List of Tables

3.1 The metrics used to monitor applications in our approach. 44

3.2 The sample applications provided by Hadoop distribution. 58

3.3 The parameters that are used to build and train our LSTM neural network. 58

3.4 Data collected from the workload simulation of Elasticsearch. 61

3.5 Data collected from the workload simulation of Hadoop applications. 61

3.6 The precision of our approach and the baselines in predicting performance anomalies. We highlight the results of the approaches that achieve the best performance. 63

3.7 The recall of our approach and the baselines in predicting performance anomalies. We highlight the results of the approaches that achieve the best performance. 64

3.8 The F-score of our approach and the baselines in predicting performance anomalies. We highlight the results of the approaches that achieve the best performance. 64

3.9 Summary of the reproduced real-world performance defects. 76

3.10 The precision of our approach and the baselines in predicting performance anomalies that are caused by the real-world performance defects. We highlight the results of the approaches that achieve the best performance. 77
3.11 The recall of our approach and the baselines. 77
3.12 The F-score of our approach and the baselines. 77
3.13 The measurements of the overhead of our approach 81

4.1 The patterns of the false positive performance defect issue reports . . 95
4.2 The results of the Wilcoxon Rank Sum tests 99
4.3 The proposed performance code metrics. The metrics are aggregated to a file level using average scheme. 113
4.4 The objects of *setForcedRefresh* method 115
4.5 The keywords that are used to identify each category of objects. The performance code metrics column represents the metrics that are calculated using each category of objects. 116
4.6 The code and process metrics used in this chapter. The last column refers to the scheme to aggregate method level metrics to a file level ("none means that no aggregation is performed for metrics that are calculated at a file level"). 118
4.7 The p-values of the Friedman tests of comparing the performance of machine learning algorithms 124
4.8 The classified groups based on the results from Nemenyi's post-hoc tests 124

4.9 The effects of groups of metrics on performance defect prediction models. We highlight the highest effects of groups of metrics in each algorithm. 129
4.10 The p-values of the Friedman tests of comparing the effects of groups of metrics 129
4.11 The classified groups based on the results from Nemenyi's post-hoc tests 129
4.12 The effects of metrics on performance defect prediction models 131
4.13 The result of the manual analysis about the types of performance defects predicted by our approach 133
4.14 The challenges in fixing performance defects 138
4.15 The result of the manual analysis on the types of performance defects that our approach fails to predict 139

5.1 The parameters that are used to fine tune CodeBERT models 159
5.2 The metrics used in this chapter. 166
5.3 The effects of groups of metrics on fixing effort prediction models. We highlight the highest effects in each algorithm. 173
5.4 The Scott-Knott ESD tests for comparing effects of groups of metrics on fixing effort prediction models 174

6.1 Decision table for highly correlated pairs 191
6.2 The number of PRs with different relevance levels. For querying the quickly merged PRs, (i) 0 indicates rejected PRs, (ii) 1 indicates slowly merged PRs and (iii) 2 indicates quickly merged PRs. For querying the quickly rejected PRs, (i) 0 indicates merged PRs, (ii) 1 indicates slowly rejected PRs and (iii) 2 indicates quickly rejected PRs. 193

6.3 An example of the ranking result of three PRs of the MovingBlocks/Terasology project for query q_1 : *quickly merged PRs* 197
6.4 The optimal rank of PRs in Table 6.3 197
6.5 The median Cliff's Delta estimate of all k positions and the Cliff's Delta magnitude 199
6.6 Example ranking top 10 PRs results of the LtR random forest model 203
6.7 Example ranking top 10 PRs results of the small-size-first baseline . . 204
6.8 Effects of each ranking metric on PRs. 205
6.9 Types of pull requests [116] 210
6.10 The number of PRs in each type 211
6.11 The standard deviation of experiences of reviewers 213
6.12 The result of Kruskal-Wallis H test 213

List of Figures

1.1 Overview of detecting and fixing performance anomalies in software systems. 6

1.2 Overview of our proposed approaches in this thesis. 10

3.1 An overview of our performance anomaly prediction approach. 43

3.2 Long Short-term Memory Cell. 45

3.3 The architecture of our approach. 46

3.4 An overview of our approach using the LSTM neural network to predict performance anomalies. 47

3.5 An overview of our data collection approach. 50

3.6 An example of the values of the monitoring metrics for the normal and anomalous cases of the monitored Elasticsearch and Hadoop applications. 51

3.7 An overview of our automatic defect injection approach. 52

3.8 The source code a Java class before and after injecting a memory leak bug. 53

3.9 A workload simulation example of applying a test suite to Elasticsearch. 56

3.10 An example of our approach making a true positive prediction. 59

3.11 An example of predicting anomalies that are caused by an infinite loop bug using the UBL baseline and our approach. Timestamps when our approach and the baseline approaches raise anomaly warnings are highlighted in red. 66
3.12 An example of predicting anomalies that are caused by a memory leak bug using the simple baseline and our approach. Timestamps when our approach and the baseline approaches raise anomaly warnings are highlighted in red. 68
3.13 Beanplots of the lead times from Elasticsearch and Hadoop experiments. The dotted line represents the overall median lead time. . . . 71
3.14 Beanplots of the lead times for predicting the performance anomalies caused by the five reproduced performance defects in Elasticsearch. The dotted line represents the overall median lead time. 78
3.15 A false positive prediction that is raised by our approach. Timestamps when our approach raises anomaly warnings are highlighted in red. . 80
3.16 The overview of implementing our approach to monitor an application. 83

4.1 An example of a performance defect report and its fixing commit. . . 94
4.2 The numbers of performance defect and non-performance defect issue reports in each project. 97
4.3 The beanplots of the characteristics of the performance defects and non-performance defects. 98
4.4 The overview of our approach. 101
4.5 The approach that is used to label the files that have performance defects 105

4.6 The number and the ratio of the files that have performance defects in the 80 GitHub projects 106
4.7 An example of performance defects that have non-intrusive fix and its resolution in project Tomcat [244] 107
4.8 An example of data corruption hang performance defects in file BenchmarkThroughput.java [16] of project Hadoop Distributed File System 108
4.9 An example of redundant traversal performance defects in file CandlestickRenderer.java [131] of project JFreeChart 110
4.10 An example of synchronization performance defects in file JobImpl.java [18] of project Hadoop MapReduce 111
4.11 The source code and AST of the *setForcedRefresh* method in Elasticsearch. 114
4.12 The AUC of machine learning algorithms for predicting performance defects at file level 124
4.13 The PR-AUC of machine learning algorithms for predicting performance defects at file level 125
4.14 The MCC of machine learning algorithms for predicting performance defects at file level 126

5.1 Time taken to solve buggy methods for the studied 106 Java projects 150
5.2 The overview of our approach. 153
5.3 Examples of a defect fixing commit and the modified method in the defect fixing commit 156
5.4 The number of commits that are collected from each experiment project. 157

5.5 The number and the ratio of the buggy methods in the 106 GitHub projects for fine tuning and evaluating CodeBERT models 158
5.6 The pipeline of using CodeBERT to predict buggy methods. 159
5.7 The performance of CodeBERT for predicting buggy methods for the 106 Java projects. 161
5.8 The number of buggy methods in training datasets of the 106 GitHub projects 162
5.9 The number of buggy methods in testing datasets of the 106 GitHub projects 163
5.10 The MCC of machine learning algorithms for predicting the fixing effort categories of the predicted buggy methods 170
5.11 The precision of machine learning algorithms for predicting the fixing effort categories of the predicted buggy methods 171
5.12 The recall of machine learning algorithms for predicting the fixing effort categories of the predicted buggy methods 172

6.1 The average number of open PRs and the average number of decisions made every day among 74 Java projects on GitHub. 180
6.2 The time taken to make decisions on PRs among 74 Java projects on GitHub. 182
6.3 The overview of our approach. 184
6.4 An overview of our ranking approach. 194
6.5 An overview of our time-sensitive evaluation 196
6.6 The performance of the LtR models to rank PRs that can be quickly merged 199

6.7 The performance of the LtR models to rank PRs that can be quickly rejected 200
6.8 The performance of our LtR approach and two baselines to rank PRs that can be quickly merged. 202
6.9 The performance of our LtR approach and two baselines to rank PRs that can be quickly rejected. 202
6.10 The performance of our LtR approach to rank PRs that can be quickly merged under three different thresholds. 208
6.11 The performance of our LtR approach to rank PRs that can be quickly rejected under three different thresholds. 209

Chapter 1

Introduction

In recent years, large-scale software systems have dramatically changed our daily activities, such as conducting business services, hosting interpersonal communication, and performing online shopping. To meet users' expectations, software systems are required to perform their functionalities with high performance and reliability. However, the performance of software systems is prone to performance anomalies [185, 192, 234]. To maintain the high performance of software systems, operators and developers are required to detect and fix performance anomalies. In the following subsections, performance anomalies, performance defects, software performance monitoring process, and pull-based development are explained and discussed.

1.1 Background

1.1.1 Performance Anomalies

Performance anomalies are unexpected performance degradation that deviates from the normal behaviors of software systems [233]. For example, the page loading process of a website can become unexpected slow or stuck if the web application server is

running of memory and disk spaces. When performance anomalies occur, the monitored system behaviors cannot be explained by the current system workload [47], e.g., the number of transactions that are processed by a system suggests less CPU and memory consumption than the actual resource usage. In production environments, performance anomalies usually manifest themselves as the violations of Service Level Objective (SLO) (e.g., long response time for users' requests) or system failures, which can frustrate users and cause financial loss [59]. For example, in August 2013, Amazon was down for 40 minutes. Amazon estimated $5 million revenue loss [247] because of the outage.

Performance anomalies can be introduced by various reasons, such as misconfiguration of the software systems, hardware failures, resource contentions, performance bottlenecks, and programming errors in the software systems [233].

1.1.2 Performance Defects

The programming errors that cause performance anomalies at run-time are referred as performance defects. Performance defects (e.g., memory leak bugs and infinite loop bugs) are non-functional defects that can significantly reduce the performance of an application (e.g., software hanging or freezing) and lead to poor user experience and waste of computational resources [185, 192, 276]. Performance defects widely exist in software systems. As reported by the existing study [139], developers in Mozilla have been fixing 560 performance defects reported by users every month over the last decade. In addition to open source software, the performance of well tested commercial products, such as Internet Explorer, Microsoft SQLServer, and Visual Studio, are affected by performance defects [106, 184, 192].

Various prior approaches [53, 86, 191, 195, 280] have been proposed to recognize performance defects at early stage of the development phase. For example, anti-patterns are design and implementation styles which lead to poor source code quality [232] and existing studies identify anti-patterns that lead to performance defects, such as data corruption hang bugs [53], redundant traversal bugs [195], memory and resource leak bugs [86], and synchronization bugs [280]. The anti-patterns (e.g., the exit condition of a loop depends on I/O operations) are used as restricted rules to check if source code contains performance defects.

1.1.3 Software Performance Monitoring Process

To maintain the performance of software systems at run-time, operators are required to detect anomalies and prevent failures from happening [137]. Operators monitor system behaviors and states by analyzing the continuously collected monitoring information, including console logs and measurement metrics (e.g., resource utilization information). Once performance anomalies are identified, operators can take actions to mitigate the impact of the anomalies. For example, if operators identify a cloud virtual machine (VM) is running out of memory, to mitigate the anomaly impact, operators can migrate the faulty VM to a physical host with more memory space [236].

The performance monitoring process requires the operators to detect anomalies on time. However, the increasing scale and complexity of software systems make it a challenging task for operators to manually identify anomalies and recover the system correctly [147]. For example, it is difficult for operators to keep track the execution status of a software system that is running on a cloud with hundreds of VMs. In addition, the large volume of monitoring data (e.g., system logs) generated during

the execution of the software systems can easily overwhelm operators. To address the challenge in manually identifying anomalies, automated system monitoring and anomaly detection using machine learning methods have been well studied by the research community [133, 137, 187, 188, 218, 253].

1.1.4 Pull-based Development

Pull-based development is a paradigm for distributed software development to bring high quality code changes from developers' workspaces to the releases of software systems. Pull-based development has been adopted in several collaborative software development platforms such as GitHub, GitLab and Bitbucket [96]. Compared with other classic distributed development approaches (e.g., sending patches to development mailing lists [30]), the pull-based development provides automated mechanisms for tracking and integrating contributions from developers [199]. For example, with the pull-based development, code changes can be integrated with just one click, without manual intervention. In this subsection, we introduce the key features in pull-based development.

Software repository. Under pull-based development, each project has a software repository, namely central repository or main repository, which stores all the code and data of the project. Only a team of core developers has the access to make changes to the central repository. However, developers can fetch clones of the central repository of any open source software or corporate software that developers can access [93]. The clones are local repositories for developers and changes made to clones are independent of each other.

Issue reports. On collaborative software development platforms(e.g., GitHub,

GitLab and Bitbucket), issue reports are reports that are submitted by developers or users to report issues (e.g., performance defects or functional errors) and request new features [142]. Developers can conduct code changes to their local repositories to fix bugs or implement new features as requested in the issue reports.

Commits. Commits are snapshots that capture the changes made by developers to one or more files in their local repositories. Each commit should be responsible for recording a single and complete work instead of bundling unrelated code changes (e.g., bug fixing and code refactoring) [61]. Developers can accumulate commits in their local repositories till they implement all the required changes to fix a bug or implement a new feature [23].

Pull requests. After conducting code changes in local repositories, developers request to have their code changes merged into the central repository through pull requests [143, 251, 271]. For example, developers can conduct code changes to correct performance defects that are responsible for performance anomalies and submit the code changes through pull request. Once developers submit pull requests, the pull requests become available to reviewers. Reviewers are the core developers that have write access to projects and are responsible for reviewing the code changes in pull requests, providing feedback, conducting tests, and requesting further changes. Finally, if a pull request is approved by reviewers, its respective code is merged into the central repository and is delivered to users on the next release of the software system.

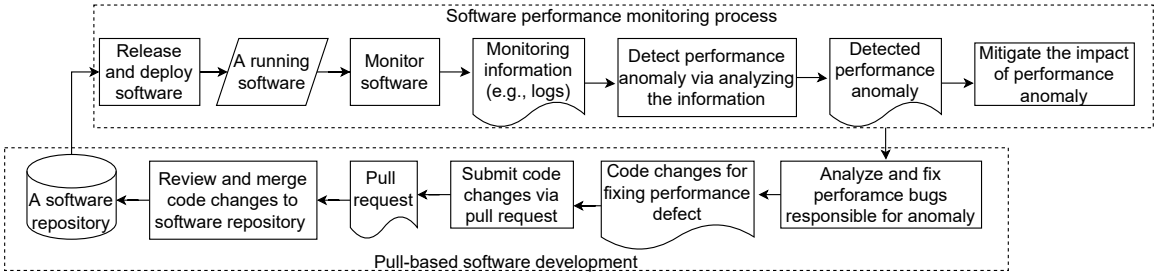


Figure 1.1: Overview of detecting and fixing performance anomalies in software systems.

1.1.5 Overview of Detecting and Fixing Performance Anomalies

Figure 1.1 shows the overall process of detecting and fixing performance anomalies in a software system. As shown in Figure 1.1, in the software performance monitoring process, after deploying a software system in production environment, the system is under monitoring. The continuously collected monitoring information (e.g., console logs and measurement metrics) are then analyzed by operators or automated anomaly detection approaches [133, 137, 187, 188, 218, 253] to detect performance anomalies. Next, corresponding actions are taken to mitigate the impact of the detected performance anomalies. To prevent the detected performance anomalies from recurring, the development and operation teams work together to find the performance defects in the source code that are responsible for the anomalies. Then, developers conduct code changes in their local repositories to fix the performance defects. Developers submit their code changes to the central repository of the software system via pull requests. Next, reviewers review the pull requests and merge the code changes to the central repository. Finally, the detected performance anomalies are fixed in the next release of the software system.

1.2 Research Challenges

Despite the existing approaches of detecting performance anomalies and performance defects and the advantages of pull-based development in integrating fixes for performance defects, the development and operation teams face the following challenges:

- **Lack of approaches to predict performance anomalies in software systems at run-time**. Various approaches [133, 137, 187, 188, 218, 253] have been proposed by the research community to detect anomalies by analyzing execution logs and resource utilization metrics. However, the detection approaches can only detect anomalies after the anomalies have happened and cannot predict the anomalies ahead of time; such limitation causes an inevitable delay in taking corrective actions to prevent performance anomalies from happening at run-time. Hence, it is desirable to provide a performance anomaly prediction approach that can proactively raise anomaly warnings in advance, and thereby help operators to prevent potential anomalies from happening.

- **Lack of general approaches to predict performance defects**. Existing studies [53, 86, 191, 195, 280] find that each type of performance defects follows a unique code-based performance anti-pattern and propose different approaches to detect such anti-patterns by analyzing the source code of a program. However, each approach can only recognize one performance anti-pattern. It is time-consuming to configure and apply different approaches separately to identify different performance anti-patterns. Moreover, prior approaches [53, 86, 191, 195, 280] cannot predict the performance defects that do not follow the identified anti-patterns. Therefore, it is important to provide a

unified approach that can predict a large variety of performance defect types.

- **Lack of ability to predict fixing effort for predicted defects**. Existing defect prediction studies [54, 108, 122, 167, 255, 256, 281] do not provide information about the estimated effort for fixing the predicted defects. As mentioned in the existing studies [28, 118, 160], predicting fixing effort for defects can help project teams prioritize defects and coordinate effort during defect triaging. Existing studies [8, 28, 88, 118, 160, 176, 261, 285] propose approaches to analyze the descriptions and attributes (e.g., reporters, severity labels, description, and affected software components) of new issue reports and estimate the effort for fixing the defects described in the issue reports. However, existing studies require information from issue reports to predict the fixing effort of defects. Therefore, the existing studies cannot provide estimation about the fixing effort for the predicted defects which have not been reported by developers. Existing defect prediction approaches [54, 108, 122, 167, 255, 256, 281] can predict hundreds of defects in a software. There are defects that can be solved quickly (i.e., within a day), while some defects can take more than a month to fix. Thus, predicting fixing effort for the predicted defects enables developers to prioritize the fixing of the predicted defects based on their working schedule.

- **Limited support to prioritize pull requests**. Under pull-based development, developers submit pull requests to fix defects (e.g., performance defects and non-performance defects) in software systems and enhance features. The role of reviewers is key to ensure the high quality of code changes and the successfully fix of defects. However, little is known about the workload (e.g, the

number of pull requests waiting to be reviewed) of reviewers. Thus, it is essential to study the workload of reviewers and propose an approach for prioritizing pull requests to help reviewers improve their productivity if the workload is overwhelming.

1.3 Thesis Statement

Performance of software systems is prone to performance anomalies. Performance defects introduced at the development phase of software systems can cause performance anomalies at run-time. Various approaches have been proposed by the research community to detect performance anomalies at run-time and performance defects at the development phase. However, there are limited approaches to predict performance anomalies and performance defects in software systems. To help developers and operators to prevent anomalies from happening, we conduct a set of studies to predict performance anomalies using run-time monitoring data and provide techniques to predict various types of performance defects at the development phase to help developers fix defects without running software systems. To this end, we perform our research along four perspectives: (1) predict performance anomalies in software systems by analyzing run-time monitoring data; (2) predict a large variety of performance defects during development phase; (3) provide a generic approach that predict the fixing effort for the predicted defects (both performance and non-performance defects); and (4) prioritize pull requests to help reviewers review code changes. The goal of this thesis is to help the development and operation teams to avoid performance anomalies at the run-time, and capture and help developers to fix performance defects at the development phase.

Summary of Thesis Statement

To prevent performance anomalies from occurring, in this thesis, we propose a set of approaches to predict performance anomalies at run-time and predict performance defects at the development phase.

1.4 Thesis Objectives

Figure 1.2 presents an overview of our proposed approaches in this thesis. To address the challenges that are listed in Section 1.2, our approaches predict performance anomalies and performance defects and provide guidance to help developers fix predicted defects.

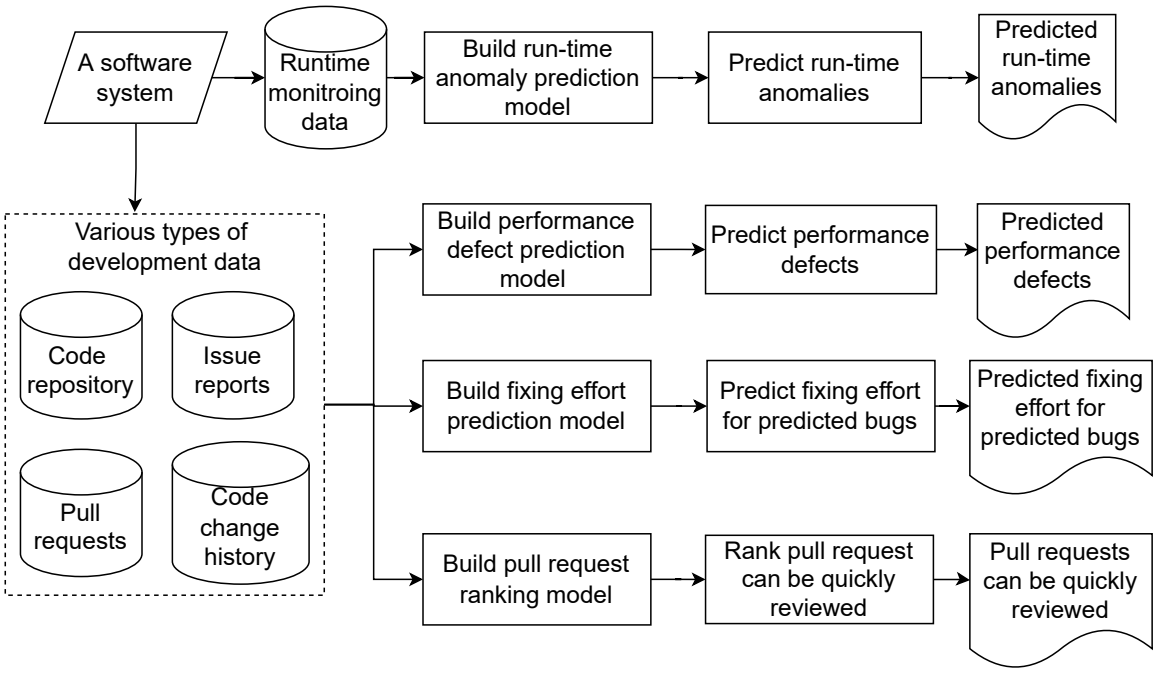


Figure 1.2: Overview of our proposed approaches in this thesis.

- **Predicting performance anomalies in software systems at run-time**.

Predicting performance anomalies in advance enables developers and operators to take proactive actions to prevent the performance anomalies from happening. We propose an approach that can predict performance anomalies in software systems and raise anomaly warnings in advance. Our approach uses a Long-Short Term Memory (LSTM) neural network to capture the normal behaviors of a software system. Then, our approach predicts performance anomalies by identifying the early deviations from the captured normal system behaviors.

- **Proposing an approach that can predict a large variety of performance defect types.** We compare the characteristics of performance defects and non-performance defects. We find that performance defects are more troublesome to fix than non-performance defects. Thus, it is important to predict performance defects to provide developers warnings at an early stage of the development phase. Next, we propose a set of performance code metrics to capture the code characteristics of performance defects. Different from the prior studies [53, 86, 191, 195, 280] that find restricted rules that match anti-patterns of performance defects, our proposed performance code metrics measure code features (e.g., the number of I/O operations and the number of loops). We build machine learning models to predict source code files that contain various types of performance defects using the proposed performance code metrics. Then, developers can prioritize their testing effort on testing the predicted buggy source code files and improve the performance of software systems.

- **Proposing a generic approach to predict methods with any types of defects (e.g., performance defects and non-performance defects) and their fixing effort**. To help developers estimate fixing effort for predicted

defects and prioritize predicted defects, we propose an approach to predict the fixing effort categories of predicted buggy methods. To evaluate the effectiveness of our approach, we use CodeBERT, a state-of-the-art language model, to predict methods that have any types of defects. The predicted buggy methods are used as the dataset for predicting fixing effort in our case studies. We measure the characteristics of predicted buggy methods via mining various history data (e.g., source code and code change history). Then, we build various machine learning models to predict if a buggy method can be fixed within one day, more than one day but within one week, or need more than one week to fix. Developers can select predicted buggy methods to fix based on their own working schedule.

- **Understanding the workload of reviewers and prioritizing pull requests**. Developers submit their changed source code of fixing defects (e.g., performance defects) and enhancing features via pull requests. We study the workload of reviewers. We find that the number of pull requests keeps on increasing over time. However, the number of reviewers and the number of pull requests that reviewers review every day remain roughly the same. To help reviewers deal with the increasing workload, we recommend pull requests that can be quickly reviewed to reviewers. Recommending reviewers pull requests that can be quickly reviewed allows them to handle more contributions and give expedite feedback on pull requests, which could be very useful when reviewers have only a limited time (e.g., half an hour or few minutes) to review pull requests. We measure the characteristics of pull requests using various types of metrics. Then, we build learning-to-rank algorithms to rank pull requests that

are likely to be quickly reviewed.

1.5 Thesis Overview

We present an overview of this thesis in the following.

- **Chapter 2 Related Work**. We give an overview of the related work of the thesis to differentiate our work from the related research.
- **Chapter 3 Predicting Performance Anomalies in Software Systems at Run-time**. We describe our unsupervised approach for predicting performance anomalies using Long-Short Term Memory (LSTM) neural network.
- **Chapter 4 Enhancing Performance Defect Prediction using Performance Code Metrics**. We describe the characteristics of performance defects compared with non-performance defects. Then, we present the proposed performance code metrics that represent the code characteristics of performance defects. We build machine learning models using the proposed performance code metrics to predict performance defects.
- **Chapter 5 Predicting Buggy Methods and Their Fixing Effort**. We present our proposed approach for predicting the fixing effort categories of the defects that are predicted by CodeBERT models.
- **Chapter 6 Adopting Learning-to-rank Algorithms for Reviewer Recommendation**. We present the workload (i.e., number of pull requests waiting to be reviewed) of reviewers, and describe our approach to help reviewers prioritize pull requests.

- **Chapter 7 Conclusions and Future Work**. In this Chapter, we conclude the thesis and discuss future directions.

Chapter 2

Related Work

The goal of this thesis is to predict performance anomalies in software systems and assist developers in fixing the predicted performance anomalies. Much research has been conducted to detect performance anomalies and defects in software systems and provide guidance information to help developers fix defects. To achieve the goal of this thesis, it is critical to understand the current research about performance anomalies detection.

In this chapter, we give an overview of the prior research related to the work conducted in this thesis across five categories: (1) performance anomaly detection; (2) performance anomaly prediction; (3) performance defect prediction; (4) predicting fixing effort of defects; and (5) improving pull request review process.

2.1 Performance Anomaly Detection

Performance anomalies are exceptional resource utilization and performance degradation issues. Performance anomalies can result in violations of Service Level Objective (SLO), e.g., slow in processing users' requests. The increasing scale and complexity of software systems make it a challenging task for operators to manually identify

performance anomalies. To automatically identify performance anomalies, automated system monitoring and anomaly detection using machine learning methods have been well studied by the research community. In this section, we present the research studies that automatically detect performance anomalies by analyzing logs and run-time metrics (e.g., resource utilization and network traffic statistics) of monitored systems.

Studies that model correlations in the monitoring data to detect performance anomalies. Munawar et al. [188] employ a linear regression model to identify correlations among monitoring metrics, such as the correlation between the number of requests to a page and the number of database connections. The measured correlations are then used to characterize normal behaviors and detect performance anomalies. Munawar et al. [187] extend their previous work [188] by comparing the use of linear regression with some of the more complex regression models, including auto-regressive regression and locally weighted regression. Guo et al. [104] apply Gaussian mixture models to detect performance anomalies. Jiang et al. [133] apply linear regressive models to characterize the flow dynamics in software systems and detect anomalies by tracking the flow dynamics. The aforementioned linear models only measure two-variable correlations in the monitoring data. In the monitoring data, the correlations might exist among multiple (e.g., three) monitoring metrics. However, searching for multiple-variable correlation can be time-consuming for software systems with many metrics (e.g., $O(n^3)$ time complexity for searching three-variable correlations for n metrics). To overcome the multi-variable correlation challenge, Jiang et al. [137] use the non-constant error variance in two-variable regression models to identify multi-variable correlations instead of searching multi-variable correlations directly.

Studies that use supervised techniques to detect performance anomalies. Cohen et al. [49] present an approach to automatically extract an indexable signature from a running system. The extracted signature represents the essential characteristic of a system state and can be used to identify whether a system state is similar to a previously anomalous state. To increase the stability of the extracted signatures, Bodik et al. [34] apply a logistic regression technique for constructing signatures from running systems. Powers et al. [206] explore different statistical techniques, including auto-regressive methods, multivariate regression methods, and Bayesian network classifiers, to detect performance violations in enterprise systems.

Studies that propose unsupervised approaches to detect performance anomalies. Shen et al. [218] propose an approach to construct profiles that characterize performance deviations between target and reference executions. Jiang et al. [136] use Normalized Mutual Information as a similarity measure to identify clusters of correlated metrics in the monitoring data. Then, the Wilcoxon Rank-Sum test is applied to identify anomalous behaviors by checking the entropy of the monitoring metrics. Similarly, Wang et al. [253] propose an entropy-based approach to detect anomalies by analyzing the distributions of the monitoring metrics. Roy et al. [215] propose a performance anomaly detection approach which can automatically discover a subset of correlated monitoring metrics and detect performance anomalies using the discovered metrics. Farshchi et al. [79] propose an unsupervised performance anomaly detection approach for Air Traffic Control systems by analyzing both log events and monitoring metrics.

Addressing the workload change challenge. In real-world software systems, the application performance changes can be introduced by workload changes instead

of performance anomalies [47]. For example, a slowdown in processing the users' requests in a web application might be introduced by the spontaneous increase in the internet traffic. It is highly desirable to avoid false alarms raised by workload changes. Stewart et al. [229] apply a transaction mix model to predict the performance of a system given a certain workload. The predicted performance can then be used to detect performance anomalies by checking if the actual performance (e.g., actual response time) deviates from the predicted performance (e.g., response time). Cherkasova et al. [47] use a regression-based transaction model with application performance signatures to distinguish between performance anomalies and workload changes. Wang et al. [257] present a clustering algorithm to detect anomalies for web applications by recognizing the workload patterns.

Addressing the data acquisition challenge. Companies are often reluctant to release their monitoring data collected from production because of the confidential issue. Therefore, datasets collected from industry settings are often too scarce for conducting research. Prior studies often collect the logs and resource utilization metrics by setting up testbed systems and simulating workloads of the systems. To obtain monitoring data from both normal and anomalous executions of a system, prior studies manually inject faults (e.g., configuration errors or programming bugs) into the system. The original system and injected system are used to collect measurement metrics from normal and anomalous executions separately. For example, Guo et al. [104] injected three performance anomalies (i.e., busy loop, deadlock, and memory leak) into an online pet store system and evaluate their approach by detecting the injected anomalies.

Studies that detect performance anomalies in cloud infrastructures.

Due to the emergence of cloud computing, many approaches are proposed to detect anomalies (e.g., machine failures) in cloud infrastructures. Bhaduri et al. [27] propose a distributed detection algorithm to detect the faulty machines in a cloud infrastructure. Fu et al. [85] propose a hybrid self-evolving anomaly detection framework using support vector machines (SVMs). Similarly, Pannu et al. [200] present a self-evolving anomaly detection framework to identify anomalies in clouds. The self-evolving approaches can continuously evolve with newly obtained monitoring data. Huang et al. [121] propose an adaptive anomaly detection approach that can be aware of the evolving behaviors of business applications on clouds. Calheiros et al. [39] propose an isolation-based anomaly detection to detect anomalies in clouds by incorporating the time-series information in the monitoring metrics. Similarly, Ibidunmoye et al. [123] propose two unsupervised techniques to detect anomalies by estimating the underlying temporal property of the time-series information in the monitoring metrics. Apart from using traditional machine learning algorithms, deep learning algorithms are studied to detect anomalies in cloud infrastructures. Gupta et al. [105] use long short term memory (LSTM) and bidirectional long short term memory (BLSTM) neural networks to detect performance anomalies.

Addressing the high search dimensionality challenge. In cloud infrastructures, there is a large number of monitoring metrics used to monitor the status of VMs. The large number of monitoring metrics can introduce high search dimensionality for training machine learning models and cause low anomaly detection accuracy [84]. To address the high dimensionality challenge, Fu et al. [84] use mutual information and principal component analysis (PCA) to select the most important metrics to reduce the search dimensionality. Similarly, Lan et al. [157] compare two techniques: PCA

and independent component analysis (ICA), for metric selection. Smith et al. [223] use Bayesian network-based dimensionality reduction to extract important features. Fu et al. [102] present a multi-scale anomaly detection approach by analyzing profiled cloud performance metrics in both time and frequency domains.

Summary

Prior research has proposed various approaches to detect performance anomalies in software system at run-time. However, prior research can only detect anomalies after the anomalies have happened and cannot predict the anomalies ahead of time; such limitation causes an inevitable delay in taking corrective actions to prevent performance anomalies from happening at run-time. In this thesis, we provide a performance anomaly prediction approach that can proactively raise anomaly warnings in advance, and thereby help operators to prevent potential anomalies from happening.

2.2 Performance Anomaly Prediction

Performance anomaly prediction approaches intend to predict performance anomalies before they occur, while performance detection approaches identify performance anomalies after the anomalies happen. In this section, we present research studies that propose both supervised and unsupervised approaches to predict performance anomalies for virtual machines (VMs) in cloud infrastructures.

Supervised performance anomaly prediction approaches. Research has explored using supervised approaches to predict performance anomalies for nodes (i.e., VMs) in cloud hosting infrastructures. Williams et al. [263] propose a supervised

black-box approach, called Tiresias, to predict failures in a distributed system by analyzing VM-level resource usage metrics collected from every node in the distributed system. Tiresias can provide a time window (i.e., lead time) for predictive system recovery. Tan et al. [234] and Gu et al. [101] combine Markov chain models and Bayesian classification models to process the VM-level resource usage metrics and predict performance anomalies. Tan et al. [235] apply a hierarchical clustering technique to discover different execution contexts in cloud infrastructures and build context-aware prediction models to improve anomaly prediction accuracy. Huang et al. [120] train Long-Short Term Memory (LSTM) neural networks to predict performance anomalies for VMs in distributed systems.

Unsupervised performance anomaly prediction approaches. Supervised approaches is limited to predict only previously known anomalies that are included in the training phase and cannot predict unknown anomalies. To predict previously unknown performance anomalies, Dean et al. [59] propose an unsupervised approach to predict the anomalies of virtual machines by analyzing the monitoring data collected from VMs in cloud infrastructures. Dean et al. [59] apply Self-Organizing Maps to learn the patterns of normal operations of all VMs in a cloud. Then, the performance anomalies are predicted by checking early deviations from the learned normal behaviors of VMs.

Addressing the overhead challenge. In real-world cloud infrastructures, the anomaly prediction approaches are required to be lightweight to avoid introducing much overhead and affecting the normal execution of the monitored VMs. To achieve low overhead, Gu et al. [100] propose adaptive data stream sampling schemes to adaptively adjust the sampling rates for collecting measurement metrics. Then,

Gu et al. [100] apply stream based classification methods to perform failure forecast in clouds.

Addressing the failure propagation challenge. In cloud infrastructures, a performance anomaly in one component can propagate to other parts of the system through the architectural dependencies and can cause a chain of errors up to the system boundary and results in a failure on the system level [20]. To predict performance anomaly before the occurrence of system level failure, Pitakrat et al. [204] propose an anomaly prediction approach that can model the failure propagation by incorporating the anomaly prediction results with the component dependencies extracted from the system architecture.

Summary

Prior research has explored unsupervised and supervised machine learning techniques and deep learning techniques to predict failures of VMs by checking the monitoring data collected from VMs in cloud infrastructures. However, the prior research has the limited capability of predicting failures at the VM-level, and operators still need to manually localize the anomalous application that causes the predicted failure in the anomalous VM. In this thesis, we provide an approach that can monitor a running application in a VM and predict performance anomaly for the monitored application.

2.3 Performance Defect Prediction

Performance defects are non-functional defects (e.g., memory leak and infinite loop bugs) in software systems that can cause performance anomalies at run-time. Different from crashing bugs or logic bugs that cause an application to stop the execution

immediately, performance defects are likely to only slow down the application. In this section, we first present the research studies that understand the life cycle of performance defects, then we present the studies that aim to predict performance defects that exist in the source code.

Understanding how performance defects are introduced. Jin et al. [139] conduct a comprehensive study of 109 real-world performance defects that are randomly sampled from five software organizations, i.e., Apache, Chrome, GCC, Mozilla, and MySQL. Jin et al. [139] find that two thirds of the studied defects are introduced by developers wrong understanding of workload or API performance features. Chen et al. [43] propose a statistical performance evaluation approach to identify performance defects in commits from Hadoop and RxJava. Then, by manually studying the issue reports that are associated with the identified performance defect introducing commits, Chen et al. [43] find that the majority of the performance defects are introduced while fixing other defects.

Understanding how performance problems are observed and reported by real-world users. Research has studied how performance problems are observed and reported by real-world users. Song et al. [226] conduct an empirical study of 65 real-world performance defects. Song et al. [226] find that users notice the symptoms of more than 80% performance defects through a comparison-based approach. Song et al. [226] also find that performance defects are reported together with two sets of inputs that look similar with each other but lead to huge performance difference. Nistor et al. [192] study performance and non-performance defects from three popular code bases: Eclipse JDT, Eclipse SWT, and Mozilla. Nistor et al. [192] find that unlike many non-performance defects, a large percentage of performance defects are

discovered through code reasoning, not through users observing the negative effects of the defects (e.g., performance degradation) or through profiling. Similarly, Ghanavati et al. [86] find that manual code inspection and manual run-time detection are the main methods for memory leak detection after conducting an empirical study on 491 memory leak bugs in Java projects. Liu et al. [171] study 70 real-world performance defects collected from large-scale and popular Android applications. Liu et al. [171] find that smartphone performance defects do not need sophisticated data inputs (e.g., a database with hundreds of entries) to manifest, but instead their manifestation needs special user interactions (e.g., certain user interaction sequences).

Understanding the patches of performance defects. Chen et al. [46] conduct a study of more than 700 performance defect fixing commits from 13 open source projects. Chen et al. [46] find that many of performance defect fixing commits follow a small set of defect patterns that are contributed by experienced developers. Similarly, Jin et al. [139] find that almost half of the examined defect patches include reusable efficiency rules that can be used to help predict and fix performance defects. In addition, Chen et al. [46] observe that the number of lines needed to fix performance defects is highly project dependent. Zhang et al. [280] conduct an empirical study on 26 synchronization performance defects in three real-world distributed systems: HDFS, Hadoop MapReduce, and HBase, to study the root cause and fix strategy of synchronization performance defects. Zhang et al. [280] find that the time consuming (e.g., nested loops) operations in critical sections are the main reason behind synchronization issues.

Predicting performance defects via software testing. Grechanik et al. [99]

propose a solution for automatically identifying performance problems in applications using black-box software testing. Jovic et al. [140] introduce an approach that can identify performance defects in an application by monitoring the behavior of the application from different deployments (e.g., application deployed on machines with different size of memory). Xiao et al. [267] suggest an approach to detect workload-dependent loops that contain time-consuming operations via monitoring the behavior of applications under large workloads. Nistor et al. [193] introduce an automated approach that reports code loops whose computations are repetitive and unnecessary. Killian et al. [149] propose an approach to predict latent performance defects by combining state-space exploration with time-based event simulation. Xu et al. [269] introduce a technique that summarizes the run-time activity in terms of data copies and identifies unnecessary operations in software systems. Coppa et al. [51] present a method for helping developers to discover hidden inefficiencies in source code. Altman et al. [6] present a tool to abstract the concrete execution states of Java applications and diagnose the root cause of idle time in applications. Han et al. [106] propose an approach that mines call-stack traces to debug the performance problems in order to identify performance defects. Pradel et al. [207] present a performance regression testing technique to test the performance of thread-safe classes. The aforementioned studies predict performance defects on running applications and analyzing their run-time information (e.g., call-stack). In contrast to the aforementioned studies, in this thesis, we propose an approach (Chapter 4) to predict performance defects in the software development process, without running the software or any test cases.

Laaber et al. [155] propose an approach to predict the stability of a performance benchmark testing using statically-computed source code features without running

the benchmark testing. Different from the existing approach [155], our approach aims to predict the performance defects in software systems without executing any test cases. Oliveira et al. [55] propose an approach to predict if a new commit affects the performance of a benchmark using the run-time monitoring information collected from running the same benchmark from previous commits. Chen et al. [44] propose prediction models to select the test cases that can manifest the performance regression in a commit. Ding et al. [62] propose an approach to predict the test cases that can demonstrate the performance improvement after fixing a performance defect. These approaches [44, 55, 62] aim to identify test cases that can manifest the existence of performance defects in software systems. The identified test cases do not reveal the exact locations of performance defects. Unlike the existing studies [44, 55, 62], our proposed performance defect prediction approach (Chapter 4) aims to identify the locations (e.g., files) of the performance defects in software systems using static source code analysis. Using our approach, developers can find the potential performance defects in source code files that may not have the functional test cases or performance test cases. After predicting source code files with performance defects, developers can introduce new test cases or modify existing test cases to test the identified files.

Predicting Performance defects via static source analysis. Jin et al. [139] use the efficiency rules that are extracted from the patches of 25 performance defects to predict performance defects in the source code. Jin et al. [139] find 332 previously unknown performance defects in MySQL, Apache, and Mozilla applications and 219 out of the 332 performance defects are found by applying rules across applications. Zhang et al. [280] propose a static tool to predict synchronization performance defects based on the common patterns of the synchronization performance

defects. Chen et al. [45] propose an automated framework to detect performance anti-patterns in Object-Relational Mapping (ORM) by analyzing global call graphs and data graphs. Nistor et al. [191] propose a static approach that detects performance defects that can be fixed by adding one line of code inside the loop. Song et al. [227] design a root-cause and fix-strategy taxonomy for real-world inefficient loops, one of the most common performance problems in the field. Then, Song et al. [227] propose a static analysis approach that can automatically predict whether a loop is inefficient based on the taxonomy. Dai et al. [53] propose a tool, called Dscope, to predict data-corruption related performance defects. Dscope analyzes I/O operations and loops in software packages and identifies loops whose exit conditions can be affected I/O operations.

The problematic usage of data structure can cause performance defects. For example, a memory leak bug can be introduced if objects added to a container data structure (e.g., an array or list) are not removed after they are no longer in use. Prior studies have proposed various approaches to predict performance defects related to problematic usage of data structures. Dufour et al. [63] present an approach to estimate the lifetimes of objects in Java applications and identify excessive usage of temporary objects. Dufour et al. [63] find that temporary data structures can include up to 12 distinct object types and can traverse through as many as 14 method invocations before being captured. Xu et al. [270] propose a static tool that can find problematic uses of container data structures by identifying the objects that are added to containers. Similarly, Bhattacharya et al. [29] present an algorithm that can predict the excessive generation of temporary container data structures within a loop by determining which objects created within the loop can be reused. Olivo et al. [195]

present a tool to identify redundant traversal performance defects, i.e., a container data structure is repeatedly iterated but the container has not been modified between successive traversals. Olivo et al. [195] apply their tool to well tested software packages, such as the Google Core Collections Library and find 72 previously unknown redundant traversal performance defects.

Summary

Prior research has proposed different approaches to predict performance defects using static source code analysis. However, each type of performance defects has a unique anti-pattern and requires a different approach to identify it. Adopting a different approach to predict each type of performance defect is time-consuming. Compared to the aforementioned studies, we propose a unified approach that can predict various types of performance defects by combining various data sources.

2.4 Predicting Fixing Effort of Defects

The estimation of fixing effort (i.e., time needed to fix a defect) is critical to help developers make decisions in coordinating effort during defect triaging. In this section, we present the research studies that analyze the descriptions and attributes (e.g., reporters, severity labels, and affected software components) of new issue reports and estimate the effort for fixing the defects described in the issue reports.

Anbalagan et al.[8] conduct an empirical study on users participation in correcting issue reports and find that there is a strong linear relationship between the number of people participating in an issue report and the time taken to fix it. The authors propose a linear model to predict the time taken to fix defects based on the number of

participants. However, it is challenging to estimate the exact number of participants for an issue report at the beginning of the fixing process. To predict the fixing time of a defect solely on an issue report, Giger et al.[88] investigate the relationships between attributes of issue reports and the time to fix issue reports. Decision tree models are built to test the relationships using six open-source projects. As a result, the decision models perform significantly better than random classification and the post-submission issue report data can improve the performance of prediction models. Hooimeijer et al.[118] classify issue reports as following (1) low effort from developers to triage and (2) high effort from developers. An approach is proposed to measure the quality of issue reports and predict whether a defect can be addressed within a given amount of time. Bhattacharya et al. [28] use multi-variate and uni-variate regression testing to test the performance of existing defect fixing time prediction models using 512,474 issue reports collected from five open-source projects. The experiment results show that the predictive power of the existing defect fixing time prediction models is between 30% and 49% and there is a need for more independent variables (features) when constructing a prediction model. In contrast to the aforementioned studies that analyze the attributes of issue reports to predict the defect fixing effort, our approach predicts fixing effort of a predicted buggy method by analyzing source code before even the method is reported as buggy.

Weiss et al.[261] predict the fixing effort of a new issue report, i.e., hours needed to fix a issue, by searching closed issue reports with similar textual descriptions. Moreover, the authors first calculate the average fixing time of the identified similar historical issue reports. The average fixing time of the similar historical issue reports are then considered as the fixing effort for the new issue report. Marks et al.[176] use

issue reports information like locations (e.g., components of the defects), reporter, and descriptions to predict their fixing time. Zhang et al.[285] use a KNN-based approach to predict whether an issue report can be fixed faster or not using attributes from issue reports and the fixing time of similar issue reports. Lee et al.[160] propose a deep learning approach to predict the fixing time of issue reports as a continuous process on log streams. Developers' activities (e.g., comments) on an issue report that are continuously logged in a bug tracking system (e.g., GitHub) are considered as log streams and are used to estimate the status of an issue report. Lamkanfi et al.[156] show that the fixing time reported in issue reports of open-source projects are heavily skewed with a significant amount of reports that are fixed in less than a few minutes. After filtering out the noisy issue reports, the performance of the defect fixing time prediction models improved. Likewise, Abdelmoez et al.[2] study the distributions of the fixing time of issue reports and identify several thresholds to filter out noisy issue reports.

Summary

Prior research has proposed various approaches to analyze the textual descriptions of issue reports and filter out noisy issue reports to improve the performance of predicting defect fixing time. However, prior studies require information from issue reports to predict the fixing effort of defects. Therefore, the existing studies cannot provide estimation about the fixing effort for the predicted defects which have not been reported by developers. In this thesis, we propose an approach that predicts the fixing effort categories of buggy methods that are predicted by a defect prediction approach. Predicting fixing effort for the predicted defects helps developers to prioritize the fixing of the predicted defects based on their working schedule.

2.5 Improving Pull Request Review Process

The pull-based development has become increasingly popular in collaborative software development platforms (e.g., GitHub, Gitlab and Bitbucket). In pull-based development model, reviewers play an important role in reviewing the code changes submitted in pull requests by developers and ensuring the high quality of code changes. In this section, we present the research studies that understand and improve the pull request review process.

Understanding the pull request review process. Rigby et al. [213] conduct an empirical study on the Apache server repository to analyze the code review practice. They study the code review process, the frequency of reviews, the interval between reviews and the quality of reviews. Jiang et al. [138] study the relationship between the accepted code changes and the time spent on code reviewing on

Linux. Gousios et al. [93] empirically study the important metrics that affect both the code review time and the code review decision on 291 GitHub projects. Steinmacher et al. [228] investigate the pull requests of casual-developers. They conducted surveys with casual-developers and reviewers to study the reason as to why pull requests from casual-developers are rejected. The authors find that the mismatch between developers' understanding and the actual developing trend of the project is among the main reason for the rejection of pull requests. Gousios et al. [98] conduct a survey with reviewers to investigate the various metrics that reviewers examine when evaluating a pull request. They observe that the match between a pull request and the current style of a project plays an important role in the code review process. Tsay et al. [246] analyze the social connection of developers in the process of evaluating contributions in GitHub. They find that pull requests from developers with a strong social connection in the project are more likely to be merged.

Recommending appropriate reviewers to review pull requests. Thongtanunam et al. [241, 242] propose a file path-based approach that represents the expertise of reviewers based on the previously reviewed file paths. Then, reviewers are ranked based on their expertise scores and the top reviewers are recommended to review the pull request. Balachandran et al. [24] introduce a reviewer recommendation approach that considers the change history of code lines to recommend reviewers. Rahman et al. [211] propose an algorithm that uses the tokens of external software libraries in a pull request to represent the pull request. Reviewers with the highest cosine similarities regarding a pull request are recommended to review the pull request. Zanjani et al. [277] introduce metrics to measure the expertise of each reviewer for each source file. Reviewers with the highest expertise scores regarding the source files

in a pull request are recommended for the pull request. Jiang et al. [135] propose an approach to measure the activeness of reviewers and recommended the most active reviewers to review pull requests.

Xia et al. [266] propose a recommendation algorithm to analyze the implicit relations in the review history. Yu et al. [272, 273] propose algorithms to capture the social connection between developers and reviewers based on the comment network. By combining the social connection and the textual similarity, their approach achieve the best performance for recommending reviewers. Similarly, Xia et al. [265] blend the file path similarity with the text similarity together to represent the expertise of reviewers. The authors test the performance of their approach on the same four open-source projects as used by Thongtanunam et al. [242]. By including the textual information, the performance improve by 8% to 61% on different projects. Jiang et al. [134] propose an approach that combines the social connection with the file path similarity to recommend reviewers to review pull requests. Van et al. [250] propose a pull request prioritization tool, called PRioritizer, to automatically recommend the top pull requests that need immediate attention of reviewers. Li et al. [166] leverage the textual similarity between incoming pull requests and other existing pull requests to warn reviewers about duplicate pull requests and prevent reviewers from wasting time on them.

Summary

Prior research has made great advances in understanding the pull request review process and recommending appropriate reviewers to review pull requests. However, the time required to review a pull request has not yet been explored when recommending a ranked list of pull requests to reviewers. This is important to quickly give feedback to developers and increase the number of pull requests that can be processed by reviewers. In this thesis, we propose learning-to-rank models to recommend quick-to-review pull requests to help reviewers to make more speedy reviews on pull requests during their limited working time.

Chapter 3

Predicting Performance Anomalies in Software Systems by Analyzing Run-time Monitoring Data

High performance is a critical factor to achieve and maintain the success of a software system. Performance anomalies represent the performance degradation issues (e.g., slowing down in system response times) of software systems at run-time. Performance anomalies can cause a dramatically negative impact on users' satisfaction. Prior studies propose different approaches to detect anomalies by analyzing execution logs and resource utilization metrics after the anomalies have happened. However, the prior detection approaches cannot predict the anomalies ahead of time; such limitation causes an inevitable delay in taking corrective actions to prevent performance anomalies from happening. We propose an approach that can *predict performance anomalies* in software systems and raise anomaly warnings in advance.

Chapter organization: Section 3.1 describes the introduction of this chapter. Section 3.2 describes the design of our approach. Section 3.3 outlines the data collection methodology. Section 3.4 presents the evaluation results for our approach. We discuss the overhead and usage scenarios of our approach in Section 3.5. The

threats to validity are discussed in Section 3.6. Section 3.7 summarizes the chapter and suggests future work.

3.1 Introduction

Large-scale software systems have become an essential part of our daily activities, such as performing financial services, fulfilling healthcare operations, and enabling interpersonal communication. To meet users' expectations, software systems need to perform their functionalities with high performance and reliability. However, the performance of software systems is prone to run-time performance anomalies (e.g., slowing down in system response time) [234]. Performance anomalies mean that the monitored system behaviors cannot be explained by the current system workload [47]. For example, the number of transactions that are processed by a system suggests less CPU and memory consumption than the actual resource usage.

To maintain the performance of software systems, operators need to detect performance anomalies and prevent failures from happening at run-time [137]. Due to the increasing scale and complexity of software systems, it is challenging for operators to manually keep track of the execution status of software systems. Hence, researchers have proposed various approaches to automatically monitor software systems and detect anomalies at run-time [133, 137, 187, 188, 218, 253].

However, the delay in detecting anomalies and taking corrective actions can result in the violations of Service Level Objective (SLO) (e.g., long response time for users' requests) or system failures, which can cause financial loss [59]. For example, in August 2013, Amazon was down for 40 minutes. Due to this outage, Amazon had almost $5 million revenue loss [247]. Hence, it is desirable to provide a performance

anomaly prediction approach that can proactively raise anomaly warnings in advance, and thereby help operators to prevent potential anomalies from happening.

Performance anomaly prediction approaches forecast that a software system is about to enter an anomalous state by capturing the pre-failure state before the anomaly happens [59]. Performance defects represent the non-functional defects, which can cause significant performance degradation [139]. Performance defects (e.g., memory leak bugs) do not always break down software systems instantly. Software systems are considered to be in a pre-failure state from the start of experiencing performance degradation until the performance anomaly occurs [59]. Prior studies propose approaches that can predict failures of virtual machines (VMs) in cloud infrastructures [59, 235]. The existing studies [59, 235] analyze the metrics collected from VMs (e.g., the CPU usage of a VM) and predict the future failures of the monitored VMs. For a cloud with hundreds of VMs, predicting VM-level failures can pinpoint the VM that will have a failure and enable operators to take proper actions to prevent the failure from happening. For example, if a VM in the cloud is running out of memory, to stop the out of memory failure from happening, operators can migrate the VM to a physical host with larger memory space [236]. However, the existing studies [59, 235] predict failures at the VM-level, and operators still need to manually localize the anomalous application that causes the predicted failure. There are multiple challenges for achieving an efficient performance anomaly prediction for applications (i.e., achieving performance anomaly prediction at the *application-level*) as follows.

- **Dynamic software behaviors**. The behaviors of software systems are dynamic (e.g., fluctuating memory usage) because of the constantly changing user

behaviors and varying system workloads. Hence, it is difficult to characterize the behaviors of a software system and precisely capture the pre-failure state [132]. Consequently, existing anomaly detection techniques that use statistical models can easily misclassify the temporal high resource utilization as anomalous behaviors [41, 173, 174, 177, 239].

- **Instrumentation challenges**. Operators do not always have access to the source code of systems running on the cloud. It is infeasible to apply existing anomaly detection techniques [25, 76] that need to instrument the source code of software systems. In addition, instrumenting the binary or source code of software systems can introduce overhead and impact the performance of the systems at run-time [76, 245].

- **Data acquisition challenges**. It is challenging to obtain enough labeled training data (i.e., monitoring data with normal and anomalous labels) to build supervised anomaly prediction approaches [101, 120, 263]. Companies are often reluctant to release their monitoring dataset as it may contain confidential information about their users. More importantly, supervised techniques can only predict known anomalies that appear in the training dataset [59].

In this chapter, we present a *performance anomaly prediction* approach that overcomes the aforementioned challenges. First, we apply a Long-Short Term Memory (LSTM) neural network to handle the difficulty in characterizing the dynamic behaviors of applications. LSTM neural network overcomes the vanishing gradient problem experienced by Recurrent Neural Network (RNN) [117]. LSTM neural network has been widely used in language modeling [231], text classification [287], and failure

prediction for physical equipment (e.g., engines [41, 173, 174, 177, 239]. Compared with other machine learning techniques (e.g., linear regression) and deep learning techniques (e.g., RNN), LSTM neural network has the advantage to incorporate the time dependency in time-series data. By incorporating the time dependency, the LSTM neural networks can capture both the temporal fluctuations and the long-term changes in the monitoring data (e.g., CPU usage) collected from an application to characterize the dynamic behaviors of the application.

To avoid the instrumentation of applications, we only use the performance monitoring application that is external to the applications to track resource utilization (e.g., the CPU and memory usage of a JAVA application). To solve the challenges related to data acquisition, we train LSTM neural networks to capture the normal behaviors of applications under regular operations. Compared with obtaining labeled data from normal and anomalous behaviors of applications, collecting only normal behaviors data is more straightforward and practical [59]. At the run-time monitoring stage, our approach predicts anomalies by checking whether the application deviates from the normal behaviors that are expected from the LSTM neural networks. Therefore, our approach apply the LSTM neural networks to predict performance anomalies without the need to train our approach using anomalous behaviors data.

Predicting performance anomalies at the application-level enables our approach to be used in multiple scenarios. Applications can be hosted on a large-scale cluster with thousands of nodes (e.g., virtual machines). Our approach can be used to monitor the instances of an application running on each node and locate the anomalous instances of the application in the cluster. In addition, in an industrial environment, there might be several applications running on the same machine or virtual machine

(VM). Operators can deploy multiple instances of our approach with each instance monitoring an application. By monitoring each application, our approach can predict which application is the anomalous application running on a machine.

The lead times measure the amount of time that our approach can predict performance anomalies in advance. After predicting a performance anomaly, operators or automatic anomaly prevention approaches [236] should take actions to prevent the predicted anomaly from happening. Anomaly prevention actions, such as scaling VM resources and live VM migration, take time to complete. Thus, our approach should predict anomalies with sufficient lead times to ensure that the anomaly prevention actions can be finished.

We conduct quantitative experiments on two different widely used open-source software systems: 1) Elatsticsearch, an open-source, distributed, and RESTful search engine [73], and 2) Hadoop MapReduce sample applications provided by Hadoop distribution [15]. Especially, our work addresses the following research questions (RQs):

RQ1. What is the performance of our approach for predicting performance anomalies?

To evaluate the performance of our approach, we inject defects that can cause performance anomalies into the source code of the studied systems and trigger the injected performance defects. Furthermore, we compare our approach with two baselines: (1) we implement the Unsupervised Behavior Learning (UBL) baseline proposed by Dean et al. [59]. (2) we implement a baseline that predicts anomalies for an application by checking if the resource utilization of the application exceeds pre-defined

thresholds. Our experiment results demonstrate that our approach can predict various performance anomalies with 97-100% precision and 80-100% recall, while the two baselines achieve 25-97% precision and 93-100% recall.

RQ2. How early our approach can raise a performance anomaly warning before the anomaly occurs?

In this RQ, we evaluate the lead times of our approach in predicting performance anomalies. Through our experiments, our approach can predict different types of performance anomalies in advance and achieve lead times varying from 20 to 1,403 seconds (i.e., 23.4 minutes). As mentioned in the existing study [236], anomaly prevention actions, such as scaling VM resources, can be completed within one second. Thus, the lead times achieved by our approach can be sufficient for the automatic prevention approaches [236] to take proper actions and prevent the anomalies from happening.

RQ3. Could our approach be used to predict anomalies happened in the real world?

In the previous RQs, we measure the performance of our approach in predicting performance anomalies that are caused by the injected defects. However, the real-world performance defects may be more complex than our injected defects. Hence, in this RQ, we demonstrate the ability of our approach for predicting performance anomalies that are caused by real-world performance defects. Because of the limited and ambiguous information in issue reports, reproducing real-world performance defects is a challenging and time-consuming task [216]. In addition, there are performance defects that can only occur under special workloads and running environments. To the end, we select and reproduce five performance defects that can be triggered

automatically through scripts in our experiment environment. Then, we use the reproduced performance defects to trigger performance anomalies. We observe that our approach can predict the anomalies that are caused by these five real-world performance defects with 95-100% precision and 87-100% recall, while the two baselines achieve 49-83% precision and 100% recall. In addition, we evaluate the run-time overhead of our anomaly prediction approach. We observe that our LSTM neural networks take an average of 0.25 milliseconds, 320MB memory, and 25% CPU usage (i.e., two CPU cores) on our machine with an 8-core Intel i7-4790 3.60GHz CPU to predict anomalies at every second.

3.2 System Design

Figure 3.1 gives an overview of our approach. As shown in Figure 3.1, our approach consists of two phases: (1) the *offline training phase* to train our LSTM neural network, and (2) the *run-time monitoring phase* to predict performance anomalies. In this section, we introduce the design of our performance anomaly prediction approach. We first describe the metrics that are used to monitor applications. Next, we introduce the architecture of our approach using the LSTM neural network. Then, we present the *training* phase of approach. Finally, we describe our approach in *predicting* performance anomaly using the trained LSTM neural network.

3.2.1 Monitoring Applications

In a real-world system, an application might co-exist with other applications. We measure the resource utilization that is specific to the application under monitoring using performance monitoring tools. Various performance monitoring tools (e.g.,

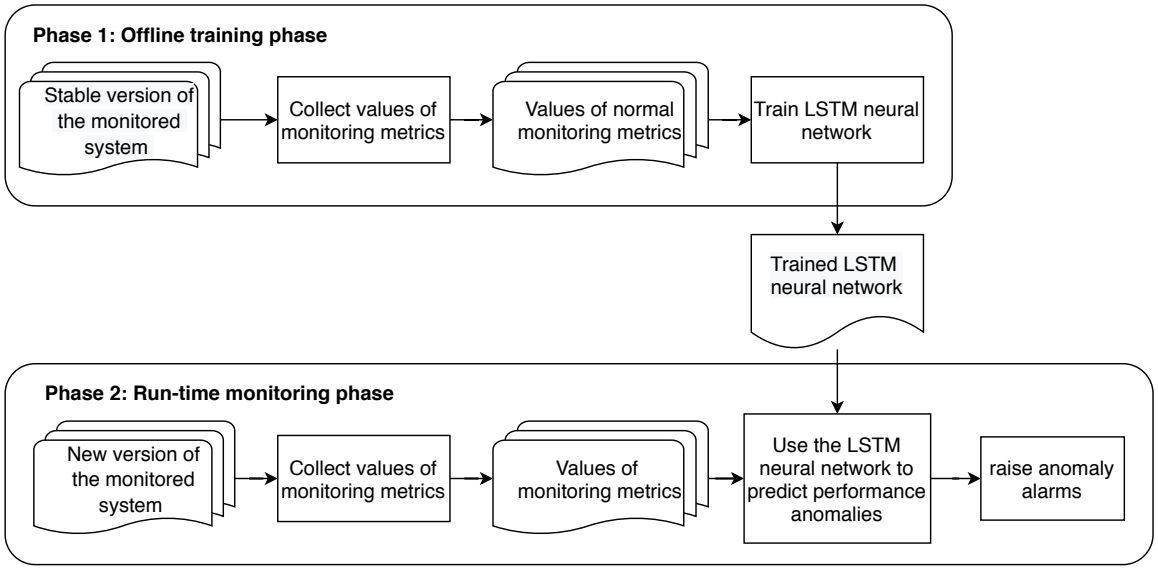


Figure 3.1: An overview of our performance anomaly prediction approach.

jconsole [198], Solarwinds [225], and tasks manager [259]) have been developed to monitor the resource utilization of an application. The performance monitoring tools can capture the resource utilization of the running processes of an application without instrumenting the application.

Table 3.1 represents the monitoring metrics that are used in our approach. We apply six metrics (e.g., CPUUsage, HeapUsage, and NativeUsage) that are commonly used to monitor the performance of application. For JAVA applications, there are four regions (i.e., eden, survivor, old generation, and code cache regions) in the heap and two regions (i.e., meta and compressed class regions) in the native memory space. The detailed descriptions of the regions can be found in the OpenJDK documentation [196]. Each region is responsible for storing a unique type of data. Hence, we monitor the usage of each region. All twelve metrics shown in Table 3.1 can be obtained using performance monitoring applications without any instrumentation.

Table 3.1: The metrics used to monitor applications in our approach.

Metric Name	Description
CPUUsage	The CPU usage of the monitored application
HeapUsage	The size of the heap of the monitored application
NativeUsage	The size of the native memory space of the monitored application
EdenUsage	The size of the eden region in the heap
SurUsage	The size of the survivor region in the heap
OldGenUsage	The size of the old generation region in the heap
CodeCacheUsage	The size of the code cache region in the heap
MetaUsage	The size of the meta region in the native memory space
CompressedClassUsage	The size of the compressed class region in the native memory space
NumThreads	The number of live threads that are issued by the monitored application
NumClasses	The number of classes of the monitored application that are loaded into memory
GcTime	The time spent in the latest garbage collection (in milliseconds)

3.2.2 Architecture of our approach

To precisely model the behavior of an application, we need to capture the time dependency of the collected metrics (i.e., the dependency between current resource usages and past resource usages). Long-Short Term Memory (LSTM) neural network is well known to accurately capture the long term time dependency and model complex correlations in time-series data (e.g., the collected values of the monitoring metrics over time) [41, 177]. The LSTM neural network is a special kind of recurrent neural network (RNN) that avoids the vanishing gradient problem existed in RNN [117]. The vanishing gradient problem means the loss of long-term dependencies introduced by

the decaying gradient values. The LSTM neural network overcomes this problem by applying multiplicative gates that enforce constant error flow through the internal states of the LSTM memory cells. The key of the LSTM neural network is the internal states of its the LSTM memory cells. Figure 3.2 shows the structure of a LSTM memory cell. There are three multiplicative gates, i.e., input gate, forget gate, and output gate in the LSTM memory cells. The input gate is used to learn what new input information should be stored in the internal states of the memory cells. The forget gate controls how long the new input information should be stored and the output gate learns what parts of the stored information the memory cells should output. These three gates prevent the internal states of memory cells from being perturbed by irrelevant inputs and outputs, which enables the LSTM neural networks to capture long term time dependency. Thus, we choose to build a predictive the LSTM neural network to model the normal behaviors of an application.

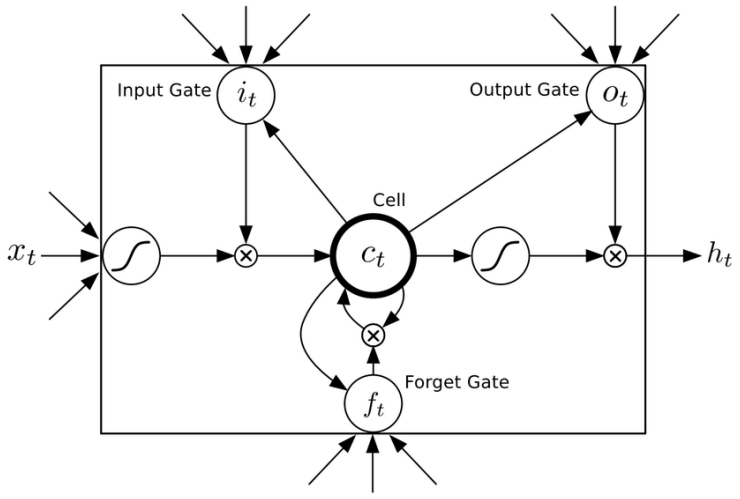


Figure 3.2: Long Short-term Memory Cell.

The architecture of our approach is shown in Figure 3.3. Our approach contains four layers: one input layer, two hidden LSTM layers, and one output layer with three

output branches. We build a deep LSTM network by stacking two LSTM layers, i.e., each neural unit in the lower LSTM hidden layer is fully connected to each unit in the higher LSTM hidden layer through feedforward connections. Prior studies show that stacking recurrent hidden layers enables a neural network to capture the structure of time-series data accurately [41, 114, 146, 173, 174, 177, 239]. Inspired by prior studies, we choose to use stacking LSTM layers to process the values of the monitoring metrics and model the normal behaviors of applications.

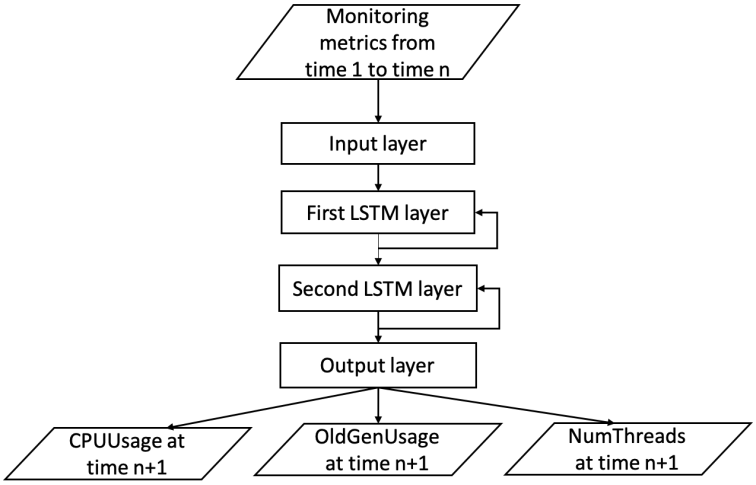


Figure 3.3: The architecture of our approach.

3.2.3 Training our Predictive LSTM Neural Network

As shown in Figure 3.4, the input to our LSTM neural network is a time series $x_1, x_2, x_3 \cdots, x_n$, where each entry $x_t, 1 \leq t \leq n$, is a 12-dimensional vector that contains the values of the monitoring metrics collected at time t. For each metric, we assign one neural unit in the input layer to take the value of the metric. To capture the behavior of an application, the next values (i.e., $x_{(n+1)}$) of the monitoring metrics should be predicted [174]. We train our LSTM neural network to predict the

CPU usage, memory usage, and the number of threads because these three metrics are common measures for the resource utilization of applications. For the memory usage, we choose to predict the $OldGenUsage$ metric that measures the size of the old generation region in the heap. More specifically, the old generation region is used to only store stable objects [154, 196]. Thus, compared with other memory usage metrics, $OldGenUsage$ contains the long-term memory usage of objects and is less fluctuating. During the training phase, our LSTM neural network tries to minimize the difference between the predicted values of the monitoring metrics and the real observed values of the monitoring metrics. We use only the values of the monitoring metrics collected from normal behaviors of applications to train our LSTM neural network. Thus, our LSTM neural network can capture normal behaviors of applications.

The output of our LSTM neural network is the predictions of $CPUUsage$, $OldGenUsage$, and $NumThreads$. The stacking LSTM layers, shown in Figure 3.3, extract the time dependency and correlations information from the input time-series values of the monitoring metrics. Then, the extracted information is passed down to the three output branches and each branch predicts the value of a monitoring metric. Different from predicting all these three metrics together in a three-dimensional vector, predicting them separately provides information loss for each metric during the training process.

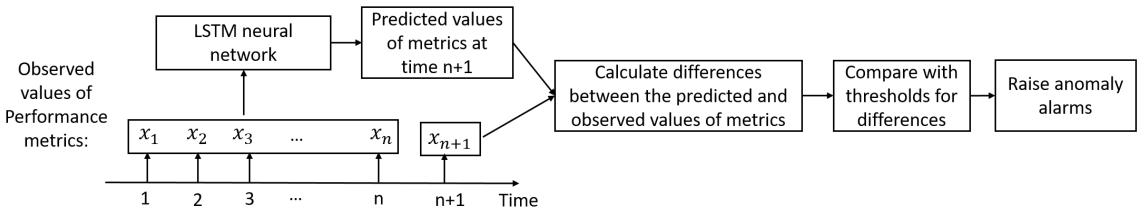


Figure 3.4: An overview of our approach using the LSTM neural network to predict performance anomalies.

3.2.4 Performance Anomaly Prediction Using the Trained LSTM Neural Network

Performance defects (e.g., memory leak and infinite loop bugs) do not always result in performance anomalies instantly. For example, it takes seven days for memory leak bug #8249 in Elasticsearch to occupy memory space and crash Elasticsearch as introduced in the issue report[1]. There is a time window, i.e., *pre-failure* time window, from the start of a performance defect until the performance anomaly occurs [59]. Therefore, we use the trained LSTM neural network to predict the future occurrences of performance anomalies by detecting whether the values of the monitoring metrics deviate from their normal behaviors.

Figure 3.4 shows an overview of our performance anomaly prediction approach. First, the trained LSTM neural network reads the collected time-series values of the monitoring metrics before time $n+1$, i.e., $x_1, x_2, x_3 \cdots, x_n$. Then, the LSTM neural network predicts the values of the monitoring metrics at time $n+1$, i.e., $\hat{CPUUsage}_{n+1}$, $\hat{OldGenUsage}_{n+1}$, and $\hat{NumThreads}_{n+1}$. Next, we calculate the differences between the three pairs of the predictions and the real observed values, i.e., ($\hat{CPUUsage}_{n+1}$, $CPUUsage_{n+1}$), ($\hat{OldGenUsage}_{n+1}$, $OldGenUsage_{n+1}$), and ($\hat{NumThreads}_{n+1}$, $NumThreads_{n+1}$). Our LSTM neural network is trained to capture the normal behaviors of an application. Thus, the prediction values are predicted following the knowledge learned from the normal behaviors. A large difference between any pairs of the predictions and the real observed values indicates that the monitored application deviates from its normal behaviors. We set up a threshold for the difference between each pair of the predictions and the real observed values. As

[1] `https://github.com/elastic/elasticsearch/issues/8249`

shown in Figure 3.4, if the difference is greater than the threshold, we think that the application is in a pre-failure state, and a performance anomaly warning is raised. Otherwise, the application is in a normal state, and no performance anomaly warning is raised. The design of the thresholds is explained in Section 3.4.1.

3.3 Data Collection

We evaluate our performance anomaly prediction approach using the data collected from two software systems: 1) Hadoop MapReduce sample applications which are JAVA open-source applications provided by Hadoop distribution [15]; and 2) Elatsticsearch which is a JAVA open-source, distributed, and RESTful search engine [73]. To test the generalizability of our approach, we conduct experiments using seven different Hadoop MapReduce applications mentioned in Section 3.3.2. The studied seven Hadoop MapReduce applications have less than 1,000 lines of code. To evaluate our approach on a large-scale software system, we test our approach on Elasticsearch which has more than 1.7 million lines of code. Elasticsearch is developed by over 1,300 developers and has more than 50,000 commits on GitHub. Furthermore, we test our approach on different released versions of Elasticsearch mentioned in Section 3.4.3.

Figure 3.5 shows an overview of our data collection approach. To train and evaluate our approach, we need to collect the metrics that track both normal and anomalous application behaviors. As shown in Figure 3.5, we first download the released versions of the studied applications. For collecting the values of the monitoring metrics for normal behaviors, we download the Elasticsearch version 5.3.0 and Hadoop applications that run on Hadoop MapReduce 3.1.3. We select these versions because these versions are well tested and widely used in practice. Next, we collect the values

of the monitoring metrics for normal behaviors by simulating the various workloads to the studied applications. Figure 3.6(a) shows an example of the values of the CPU usage for normal behaviors that are collected from Hadoop applications. Then, we inject defects into the source code of the studied applications and obtain the buggy applications. Finally, we collect the values of the monitoring metrics for anomalous cases by simulating the workloads to the buggy applications. Figure 3.6(b) shows an example of the values of the CPU usage for anomalous cases that are collected from a buggy Elasticsearch with an injected infinite loop bug. As shown in Figure 3.6(b), after the occurrence of the infinite loop bug at the time mark of 1,500 second, the CPU usage of Elasticsearch starts increasing and reaches to 100%. We explain the defect injection and workload simulation steps as follows.

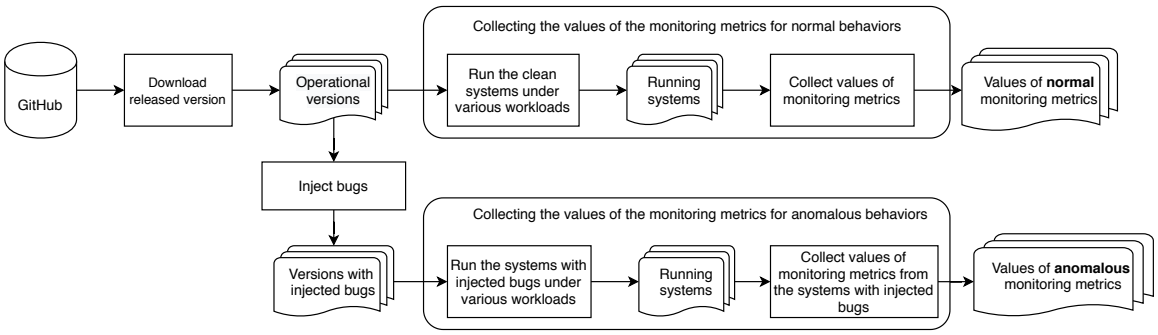
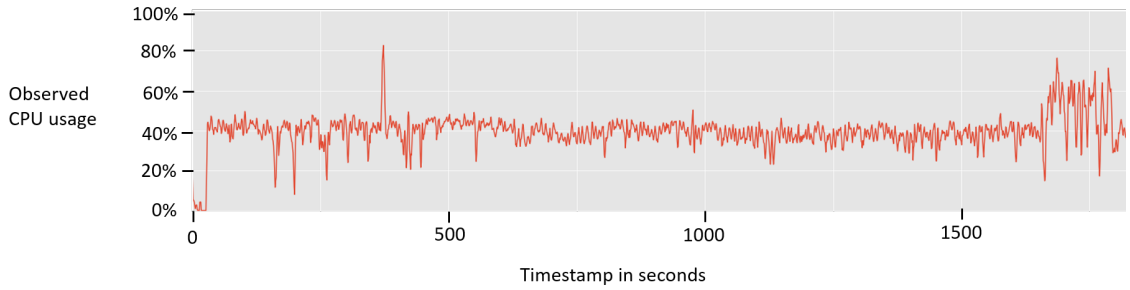


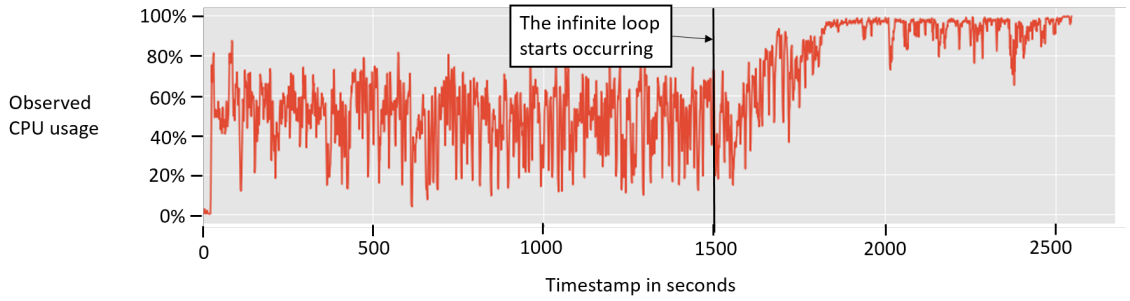
Figure 3.5: An overview of our data collection approach.

Defect injection: To simulate performance anomalies, previous studies inject defects that are related to the CPU and memory usage into the source code [59, 101, 235, 263]. Inspired by the previous studies, we select and inject three popular types of performance defects as follows.

- **Infinite loop**. We add infinite loop bugs that increase CPU consumption continuously by creating threads with infinite loops.



(a) Normal CPU usage of the monitored Hadoop applications.



(b) Anomalous CPU usage of the monitored Elasticsearch.

Figure 3.6: An example of the values of the monitoring metrics for the normal and anomalous cases of the monitored Elasticsearch and Hadoop applications.

- **MemLeak**. We inject memory leak bugs which continuously create objects and store them into static lists. Because the elements in static lists are not removed by the garbage collection process, the memory usage of the monitored JVM can keep on increasing.

- **Deadlock**. We insert buggy code to create threads with deadlock issues. The created threats deadlock each other.

We aim to inject the three types of performance defects into different source code locations of the studied applications and obtain different buggy applications. Manually injecting performance defects is error-prone and time-consuming. Therefore, we propose an automatic defect injection approach that injects defects into the studied

applications, as shown in Figure 3.7. We first select the JAVA classes in the source code to inject defects. We only select JAVA classes that can be reached during runtime to inject performance defects. Because the architectures of Hadoop MapReduce applications and Elasticsearch are different, we apply different strategies to select JAVA classes in the studies applications, as shown in Sections 3.3.1 and 3.3.2. We build the Abstract Syntax Trees (ASTs) of the selected classes which represent the syntax structures of the source code of the classes. For each selected class, we analyze the AST and identify the location to inject performance defects. Next, we inject the three types of performance defects in the source code of the selected class. Figure 3.8(a) shows the source code of a Java class in Elasticsearch. Our approach analyzes the syntax structure of the class and identifies that the definitions of the *RestAnalyzeAction* class and the *prepareRequest* function starts at lines 41 and 78, respectively. Figure 3.8(b) shows the source code of the Java class after injecting a memory leak bug. A static list is injected in the *RestAnalyzeAction* class (i.e., line 42) and a for loop is injected at the beginning of the *prepareRequest* function (i.e., line 80) to add elements into the static list.

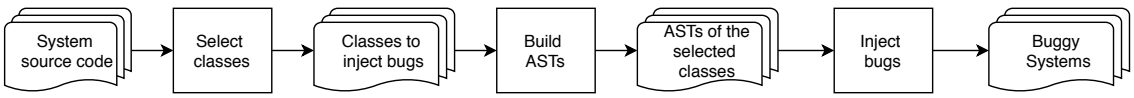


Figure 3.7: An overview of our automatic defect injection approach.

Running the normal/buggy applications under various workloads: We conduct all the workload simulation on a single machine with an 8-core Intel i7-4790 3.60GHz CPU, 32GB memory, and Ubuntu version 16.04.1. We deploy Elasticsearch and Hadoop MapReduce framework in a single-node cluster. We realize that our testbed environment is much simpler than the industrial environment in which a

```
41  public class RestAnalyzeAction {
            ...
78          public RestChannelConsumer
    prepareRequest(...) {
            ...
114         }
    ...
219 }
```

(a) The Java class before defect injection.

```
41  public class RestAnalyzeAction {
42          public static final ArrayList<Double>
    list = new ArrayList<Double>();
            ...
79          public RestChannelConsumer
    prepareRequest(...){
80          for (int i = 0; i< 300; i++){
            list.add(java.util.Random.nextDouble());
                }
            ...
114         }
    ...
219 }
```

(b) The Java class after defect injection.

Figure 3.8: The source code a Java class before and after injecting a memory leak bug.

cluster could host thousands of nodes. As our approach works on the application-level resource usage metrics, a single-node cluster in our experiment design can provide the same infrastructure as a multiple-node cluster for collecting the application-level metrics. We discuss how to use our approach in a large-scale cluster in the Discussion Section.

To demonstrate that our approach can be used in practice, we drive the subjected applications using various workloads that are observed in real-world scenarios as follows.

- For Elaticsearch, we use the datasets that are designed to conduct the macrobenchmarking test as the input workload. The macrobenchmarking test is a testing methodology used to evaluate the overall performance of applications [169].

- As shown in Table 3.2, the studied Hadoop MapReduce applications produce statistics about a set of text files. For example, the application wordMean counts the average length of the words in the input files. RandomTextWriter [14]

is a text generation application that is provided by Hadoop to generate random large text files (e.g., files with 1GB size). Hence, we generate the input datasets for the studied Hadoop MapReduce applications using the RandomTextWriter application.

Our approach collects the values of the monitoring metrics of the studied applications using a toolkit, called IBM Javametrics [124] that monitors resource usage of a JVM. IBM Javametrics can be configured to automatically store the values of the monitoring metrics, while other performance monitoring tools (e.g., tasks manager [259], or jconsole [198]) require manual effort to output the values of the monitoring metrics. We configure the JVM configurations of the studies applications to load the IBM Javametrics when the applications start running. Then, the loaded IBM Javametrics collect the resource usage of the applications at run-time with a configured sampling interval. We set the sampling time interval to one second in our experiments as used in the existing studies [59]. In the next sections, we describe the details for data collection in the studied applications.

3.3.1 Data Collection for Elasticsearch

We follow the defect injection approach shown in Figure 3.7 to obtain buggy versions of Elasticsearch. To automatically trigger the injected defects, we select the classes that handle HTTP requests. In Elasticsearch, the classes that handle HTTP requests act as entry points for processing HTTP requests. Hence, we can inject the performance defects into the classes that handle HTTP requests and trigger the injected defects once we send HTTP requests. For example, we obtain a buggy Elasticsearch by injecting an infinite loop bug into the class, *RestClusterStateAction.java*, which

handles the HTTP requests for checking the cluster states. The injected defect creates a CPU consumption thread every time when the application handles a request for checking cluster state.

The API method, *controller.registerHandler()*, is used to register the classes for handling HTTP requests. Therefore, we find all the classes that handle HTTP requests by searching the keyword, *controller.registerHandler()*. We then manually check the classes identified by searching the keyword, *controller.registerHandler()*, and filter the classes that do not process HTTP requests. After the filtering, 90 classes that handle HTTP requests are identified. For each class, we obtain three buggy versions of Elasticsearch by injecting the three different types of performance defects. In total, we obtain 90*3 (i.e., 270) buggy versions of Elasticsearch. After adding the one normal version of Elasticsearch, we have 90*3+1 (i.e., 271) versions of Elasticsearch in total.

To simulate the real-world workloads, we use Rally [72] benchmark framework. Rally is a macrobenchmarking framework for Elasticsearch, and it provides various macrobenchmarking test suites based on different datasets, such as questions and answers datasets from StackOverflow and the HTTP server log data dataset. In our experiment, a workload simulation is the execution of Elasticsearch under a macrobenchmarking test suite. Figure 3.9 shows an example of applying a test suite to Elasticsearch. As shown in Figure 3.9, the macrobenchmarking test suites simulate real-world scenarios of using Elasticsearch, such as executing term searching operations that search for different keywords or executing range searching operations that search for all documents created in a specific time window. Each test suite takes the numbers of different types of operations as configurable parameters. Hence, to

be sure that the training data and testing data for our approach are collected under different workloads, we randomize the parameters of a test suite before applying it. Then, the different types of operations are executed sequentially. After executing the test suite, we restart the Elasticsearch and start applying the next test suite. We have 271 versions (i.e., 270 buggy versions and one normal version) of Elasticsearch. We find three stable test suites that can be applied to simulate workload. Thus, for each version of Elasticsearch (e.g., normal or buggy versions), we apply these three test suites in Rally. In total, we run 271*3 (i.e., 813) test suites. The duration of each test suite varies from 45 minutes to 2 hours. It takes our machine around two months to finish running all the test suites and collecting the values of the monitoring metrics.

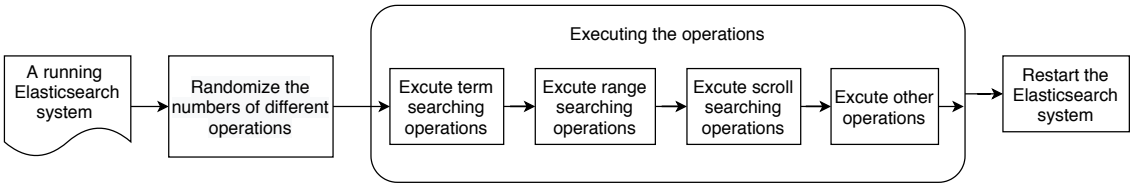


Figure 3.9: A workload simulation example of applying a test suite to Elasticsearch.

To generate data for anomalous situations, we keep on sending the HTTP requests that are handled by the injected classes to trigger the injected defects. The starting time for sending the HTTP requests and the time intervals between every two requests are randomly selected and recorded. While simulating the workloads, we track the response times of Elasticsearch for processing the submitted HTTP requests. A performance anomaly is marked if the average response time is greater than a threshold (e.g., 100ms) [59]. The marked performance anomalies are then used to evaluate the performance of our anomaly prediction approach.

3.3.2 Data Collection for Hadoop

We conduct experiments using seven sample applications that are provided by the Hadoop distribution as listed in Table 3.2. For each sample application, we use the approach shown in Figure 3.7 to automatically inject the three types of performance defects into the Map classes. Then, the injected defects are triggered automatically once the Map classes start to process input data. We have seven Hadoop applications, meaning that we obtain 7*3 (i.e., 21) buggy Hadoop applications and seven normal Hadoop applications in total. We then collect the values of the monitoring metrics following the data generation approach shown in Figure 3.5. For Hadoop applications (i.e., multiFileWc, wordCount, wordMean, wordMedian, and wordStandardDeviation), each simulated workload is 10GB text files. For Hadoop applications bbp and pi, each simulated workload is the number of the digits of π that needs to be calculated. To ensure that the training data and testing data for our approach are collected under different workloads, we generate different text files and the number of the digits of π whenever applying them to Hadoop applications.

We run the normal and buggy versions of these seven applications to collect the values of the monitoring metrics under normal and anomalous situations. We program the injected performance defects to record their start time at run-time. We monitor the progress of each application and mark a performance anomaly when the application does not make any progress for processing the input data.

3.4 Evaluation

In this section, we present the approach and the results of the studied research questions.

Table 3.2: The sample applications provided by Hadoop distribution.

Application	Description
bbp	An application that uses Bailey-Borwein-Plouffe algorithm to compute the exact digits of Pi.
pi	An application that estimates Pi using a quasi-Monte Carlo method.
multiFileWc	An application that counts the words in several input files separately.
wordCount	An application that counts the words in the input files.
wordMean	An application that counts the average length of the words in the input files.
wordMedian	An application that counts the median length of the words in the input files.
wordStandardDeviation	An application that counts the standard deviation of the length of the words in the input files.

3.4.1 RQ1. What is the performance of our approach for predicting performance anomalies?

Motivation: To test the ability of our approach for predicting performance anomalies, in this RQ, we measure the performance of our approach and compare it with the existing state-to-the-art approaches.

Table 3.3: The parameters that are used to build and train our LSTM neural network.

Parameter	Value
Batch size	8
Number of input features	12
Number of steps	4
Number of hidden cells in LSTM layers	8
Number of epoches	20
Optimization function	Adam [151]
Loss function	Mean square error [264]

Approach: We collect the values of the monitoring metrics under normal and anomalous situations from the studied applications, as described in Section 3.3. We build our LSTM neural network using Keras [148]. Table 3.3 lists the parameters that are used to build and train our LSTM neural network. The detailed descriptions of the parameters can be found in the Keras documentation [148]. We train our LSTM neural network using the values of the monitoring metrics under normal behaviors, as introduced in Section 3.2.3. In the run-time monitoring phase, our LSTM neural network predicts the next values of the monitoring metrics by analyzing the current and past values of the monitoring metrics. In particular, our approach predicts the values for three monitoring metrics (i.e., $CPUUsage, OldGenUsage$, and $NumThreads$). Finally, we compare the predicted values with the actual observed values and raise anomaly warnings if the differences are larger than the predefined thresholds.

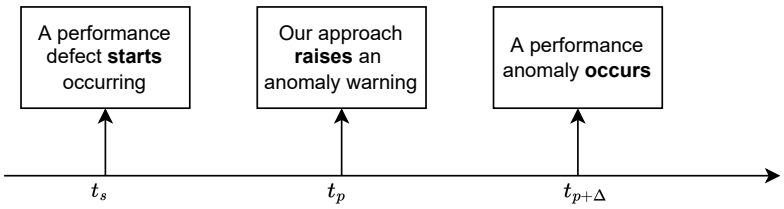


Figure 3.10: An example of our approach making a true positive prediction.

Measuring the performance of our approach. We use precision and recall to measure the performance of our approach for predicting performance anomalies. As shown in Figure 3.10, we consider that our approach achieves a *true positive* prediction if our approach raises an anomaly warning after the starting of the performance defect and before the occurrence of the performance anomaly. If our approach raises a warning, but there is no performance defect happening, we consider that our approach makes a *false positive* prediction. If our approach does not raise any warning, but

the performance anomaly actually occurs, we consider that our approach *fails* to predict the performance anomaly. Equations 3.1 and 3.2 show the computation for precision and recall. Precision measures the percentage of *true positive* predictions among all the predictions made by our approach. Recall represents the percentage of the performance anomalies that can be predicted by our approach among all the anomalies in our experiment datasets.

$$Precision = \frac{Num_{tp}}{Num_{tp} + Num_{fp}} \tag{3.1}$$

$$Recall = \frac{Num_{tp}}{Num_{tp} + Num_{fn}} \tag{3.2}$$

Where Num_{tp} and Num_{fp} are the number of true positive (TP) and false positive (FP) predictions that our approach makes. In addition, Num_{fn} is the number of false negatives (FN), i.e., the number of performance anomalies that our approach fails to predict.

Identifying the optimal thresholds for raising performance anomaly predictions. We use the values of the monitoring metrics under normal situations as our training datasets. We split the data for the values of the monitoring metrics under anomalous situations into the validation and testing datasets as shown in Tables 3.4 and 3.5. There are more buggy versions of the studied applications than the normal versions. Therefore, there are more data in the validation and testing datasets than the data in the training datasets. As mentioned in Section 3.3, we simulate different workloads to collect data. Thus, the training and testing data shown in Tables 3.4 and 3.5 are collected under different workloads, which enables us to measure the performance of our approach under different workloads from the training phase.

Table 3.4: Data collected from the workload simulation of Elasticsearch.

Datasets	**Description**	**Number of collected samples**
Training	The values of the monitoring metrics that are collected by running the normal version of Elasticsearch.	45,801
Validation	30% of the values of the monitoring metrics that are collected by running the buggy versions of Elasticsearch.	199,973
Testing	70% of the values of the monitoring metrics that are collected by running the buggy versions of Elasticsearch.	466,603

Table 3.5: Data collected from the workload simulation of Hadoop applications.

Datasets	**Description**	**Number of collected samples**
Training	The values of the monitoring metrics that are collected by running all normal versions of the Hadoop applications.	36,625
Validation	30% of the values of the monitoring metrics that are collected by running the buggy versions of the Hadoop applications.	19,989
Testing	70% of the values of the monitoring metrics that are collected by running the buggy versions of the Hadoop applications.	46,641

We record the time taken to train the LSTM neural networks on our experiment machine with an 8-core Intel i7-4790 3.60GHz CPU and 32GB memory. It takes 276 seconds (i.e., 4.6 minutes) and 240 seconds (i.e., 4 minutes) to train the LSTM neural networks for Elasticsearch and Hadoop applications, respectively. We set up three thresholds for the values of the three monitoring metrics predicted by our LSTM

neural network (i.e., $CPUUsage, OldGenUsage$, and $NumThreads$). Similar to the existing studies [41], we choose the thresholds by conducting sensitivity analysis to find the values that can yield to the maximum F-score on the validation datasets. F-score is a combination metric of precision and recall. Equation 3.3 shows the computation for F-score.

$$F - score = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{3.3}$$

We choose the thresholds using the validation dataset. Then, the selected thresholds are used on the testing datasets to predict performance anomalies and evaluate the performance of our approach.

Comparing with the existing state-to-the-art approaches. We select the Unsupervised Behavior Learning (UBL) approach proposed by Dean et al. [59] as a baseline. The UBL baseline is unsupervised and is capable to predict performance anomalies by analyzing the values of the monitoring metrics. UBL baseline uses the Self Organizing Map (SOM) to capture the behaviors of applications. We build the SOM using the minisom python library [141]. For the configuration of the SOM, we use the best configuration presented by Dean et al. [59] in their experiments. We randomly initialize the weight vectors of the SOM as used in the UBL baseline [59]. We train the SOMs on our training datasets listed in Tables 3.4 and 3.5. The UBL baseline needs a threshold to differentiate the normal and anomalous values of the monitoring metrics. We determine the threshold by conducting the sensitivity analysis on the F-score using our validation datasets. Finally, we test the performance of the UBL baseline on our testing datasets.

In addition to the UBL baseline, we implement a simple baseline that raises performance anomaly warnings for an application by checking if the resource utilization of the application exceeds pre-defined thresholds. For example, VMware [2], one of the top companies in providing cloud computing and virtualization software and services, suggests raising anomaly warnings once the CPU usage of a virtual machine (VM) is above 95% [252]. We set up the thresholds for the values of the three monitoring metrics, i.e., $CPUUsage$, $OldGenUsage$, and $NumThreads$ and predict performance anomalies by checking if any of the values of these three metrics exceeds its threshold. We determine the thresholds for the values of these three monitoring metrics by maximizing the F-score on our validation datasets listed in Tables 3.4 and 3.5. We compare the performance of our approach with the performance of the two baselines (i.e., the UBL and the simple baselines).

Table 3.6: The precision of our approach and the baselines in predicting performance anomalies. We highlight the results of the approaches that achieve the best performance.

	Elasticsearch			Hadoop applications		
	Deadlock	Infinite loop	Memory leak	Deadlock	Infinite loop	Memory leak
Our approach	**98%**	**97%**	**100%**	**100%**	**100%**	**100%**
The UBL baseline	97%	65%	25%	41%	85%	70%
The simple baseline	76%	79%	78%	67%	60%	71%

Results: Our approach can predict various performance anomalies with 97-100% precision and 80-100% recall. As shown in Tables 3.6, 3.7, and 3.8, for **Elasticsearch**, our approach is able to achieve high precision of 98%, 97%, and

[2] `https://www.vmware.com/`

Table 3.7: The recall of our approach and the baselines in predicting performance anomalies. We highlight the results of the approaches that achieve the best performance.

	Elasticsearch			Hadoop applications		
	Deadlock	Infinite loop	Memory leak	Deadlock	Infinite loop	Memory leak
Our approach	98%	**100%**	87%	**100%**	**100%**	80%
The UBL baseline	**100%**	93%	**94%**	**100%**	**100%**	**100%**
The simple baseline	**100%**	**100%**	93%	**100%**	**100%**	**100%**

Table 3.8: The F-score of our approach and the baselines in predicting performance anomalies. We highlight the results of the approaches that achieve the best performance.

	Elasticsearch			Hadoop applications		
	Deadlock	Infinite loop	Memory leak	Deadlock	Infinite loop	Memory leak
Our approach	**98%**	**98%**	**93%**	**100%**	**100%**	**89%**
The UBL baseline	**98%**	77%	39%	58%	92%	82%
The simple baseline	86%	88%	85%	80%	75%	83%

100% in predicting the performance anomalies that are caused by the deadlock, infinite loop, and memory leak bugs, respectively. For **Elasticsearch**, our approach achieves the highest recall of 100% and 98% in predicting the performance anomalies that are caused by the infinite loop and deadlock bugs, respectively. For **Hadoop applications**, our approach achieves 100% precision and 100% recall in predicting the performance anomalies that are caused by the infinite loop and deadlock bugs.

The good performance regarding the infinite loop and deadlock bugs suggests that our approach models the behaviors of the normal CPU usage and the number of

threads accurately. Conversely, We find that the lowest recall (i.e., 80% and 87%) of our approach occurs when the performance anomalies are caused by the memory leak bugs. The low performance for predicting the anomalies caused by the memory leak bugs can be explained by the high fluctuation of the memory usage. Memory usage depends on the size of the data that are processed by the applications. Because we run the studied applications to process large datasets (e.g., 10GB text files for the Hadoop MapReduce applications), the memory usage of the applications is fluctuating. The high fluctuation of the memory usage impacts the ability of our approach to distinguish between the normal and anomalous behaviors of memory usage. Consequently, the high fluctuation reduces the recall of our approach in predicting the performance anomalies that are caused by the memory leak bugs.

Our approach consistently outperforms the UBL and the simple baselines in predicting performance anomalies in Elasticsearch and Hadoop applications. We highlight the results of the approaches that achieve the best performance in Tables 3.6, 3.7, and 3.8. Our approach consistently achieves higher precision and F-score than the baselines. For **Elasticsearch**, our approach achieves an average precision of 98.3% in predicting anomalies that are caused by different types of performance defects, while the UBL and the simple baselines achieve an average precision of 62.3% and 77.7%, respectively. For **Hadoop applications**, both our approach and the UBL baseline achieve high precision and recall in predicting the performance anomalies that are caused by the infinite loop bugs. The reason behind the high performance is that the CPU usage of the normal Hadoop applications does not experience high fluctuation during the execution of the map tasks. The map tasks execute the same operations on different chunks of data over time. The simple

baseline achieves a low precision of 60% in predicting anomalies that are caused by the infinite loop bugs. One reason for the low performance is that the CPU usage threshold learned from the Hadoop validation dataset cannot scale to the CPU usage behaviors in the Hadoop testing dataset.

Anomaly prediction examples. The aforementioned results show that our approach outperforms the baselines, especially, our approach achieves higher precision in predicting performance anomalies. To obtain insights about the high precision of our approach, we provide examples to show the cases where the baselines raise false positive predictions while our approach does not.

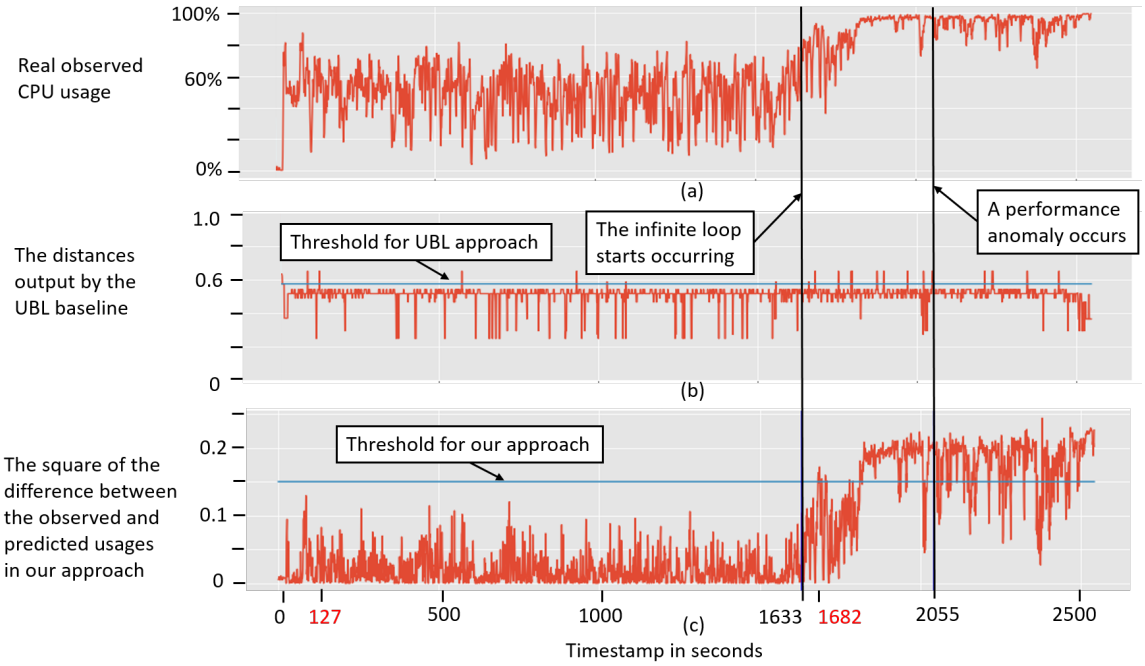


Figure 3.11: An example of predicting anomalies that are caused by an infinite loop bug using the UBL baseline and our approach. Timestamps when our approach and the baseline approaches raise anomaly warnings are highlighted in red.

Figure 3.11 shows an example of using our approach and the UBL baseline to

predict a performance anomaly that is caused by an infinite loop bug. As proposed by Dean et al. [59], for the values of the monitoring metrics collected at each second, the UBL baseline maps the values of the metrics into a winner neuron in the Self Organizing Map and calculates the distance between the winner neuron and its neighbor neurons. The distances for the collected values of the monitoring metrics are shown in Figure 3.11 (b). If the distance is greater than a pre-defined threshold, a performance anomaly warning is raised. As mentioned in Section 3.2, our approach raises performance anomaly warnings by checking the differences between the predicted values of the monitoring metrics and the real observed values of the metrics. The differences between the predicted and real observed values are shown in Figure 3.11 (c). If the difference is greater than a threshold, our approach raises a performance anomaly warning. As introduced in Section 3.4.1, the thresholds for our approach and the UBL baseline are selected by maximizing the F-score on the Elasticsearch validation dataset shown in Table 3.4.

As shown in Figure 3.11, the CPU usage of Elasticsearch keeps on fluctuating from the beginning because of varying operations in the workload simulation. After an infinite loop bug starts to occur at the time mark of 1,633 second, the real observed CPU usage continues to increase and reaches to 100%. Both our approach and the UBL baseline make a true positive anomaly prediction at the time mark of 1,682 second. However, the UBL baseline raises several false positive predictions before the occurrence of the infinite loop bug because of the fluctuation of the CPU usage. For example, at the time mark of 127 second, the real CPU usage of Elasticsearch has a large fluctuation and the UBL baseline raises a false positive prediction. Compared with the UBL baseline, no false positive predictions are raised by our approach.

Our LSTM neural network is not impacted by the fluctuations as the LSTM neural network can capture both the short fluctuations and the long-term changes.

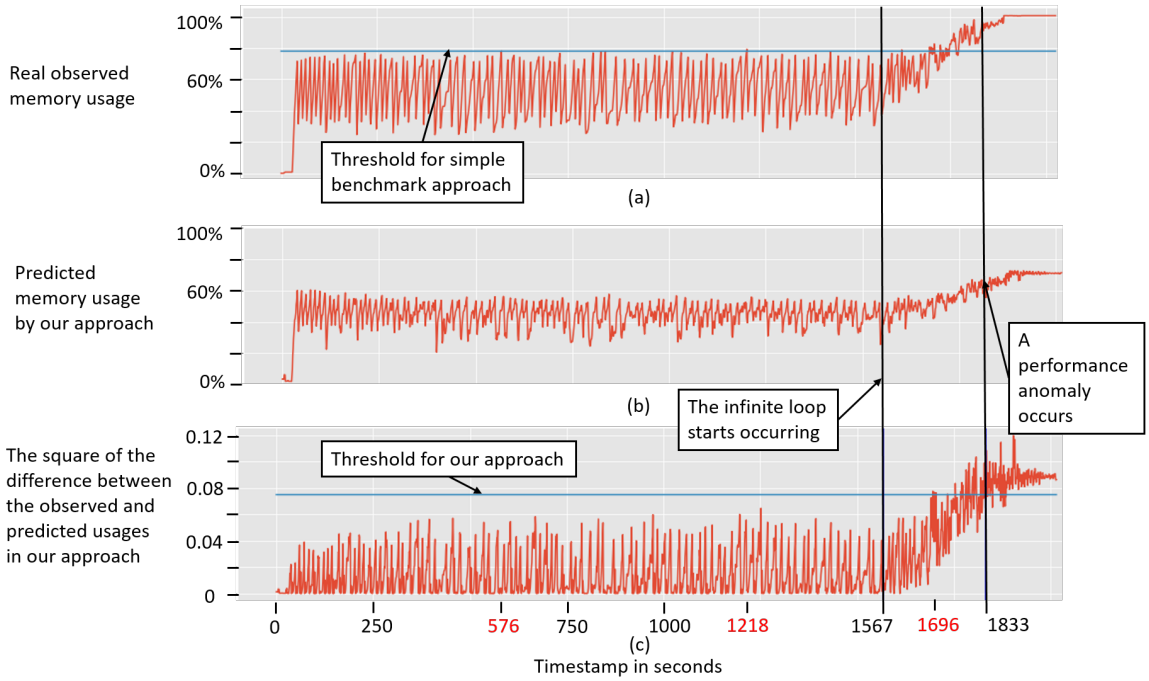


Figure 3.12: An example of predicting anomalies that are caused by a memory leak bug using the simple baseline and our approach. Timestamps when our approach and the baseline approaches raise anomaly warnings are highlighted in red.

Figure 3.12 shows an example of using our approach and the simple baseline to predict a performance anomaly that is caused by a memory leak bug. After a memory leak bug starts happening at the time mark of 1,567 second, the real observed memory usage of Elasticsearch continues to increase and reaches to 100% of the memory space. As shown in Figure 3.12, both our approach and the simple baseline are able to predict the performance anomaly around the time mark of 1,696 second. However, the simple baseline raises several false positive predictions before the occurrence of the memory leak bug. The memory usage threshold learned from the validation dataset is 78%. At

the time marks of 576 and 1,218 seconds, the memory usage of Elasticsearch increases and exceeds the threshold. Compared with the simple baseline, our approach is more robust to the fluctuations of the memory usage and achieves a higher precision.

The reasons behind the good performance of our approach. The aforementioned results and examples show that our approach achieves good performance in predicting performance anomalies. The explanation for the good performance of our approach is that the LSTM neural networks can incorporate the time dependency in time-series data (e.g., the values of the monitoring metrics in our case). In addition, we use two stacking LSTM layers that have been observed to capture the time dependency accurately in the existing work [41, 173, 174, 177, 239].

From the aforementioned examples, we demonstrate that the LSTM neural networks can predict anomalies more accurately than the UBL baseline. We find that the LSTM neural networks are not impacted by the fluctuations in the values of the monitoring metrics as the neural networks can capture both the short fluctuations and the long-term changes. For example, the CPU usage of Elasticsearch has high fluctuations because of varying workloads. The high fluctuations result in the low precision of the UBL baseline in predicting anomalies (e.g., 25% precision in predicting anomalies caused by the memory leak bugs), while our approach achieves a precision of 100%.

Summary

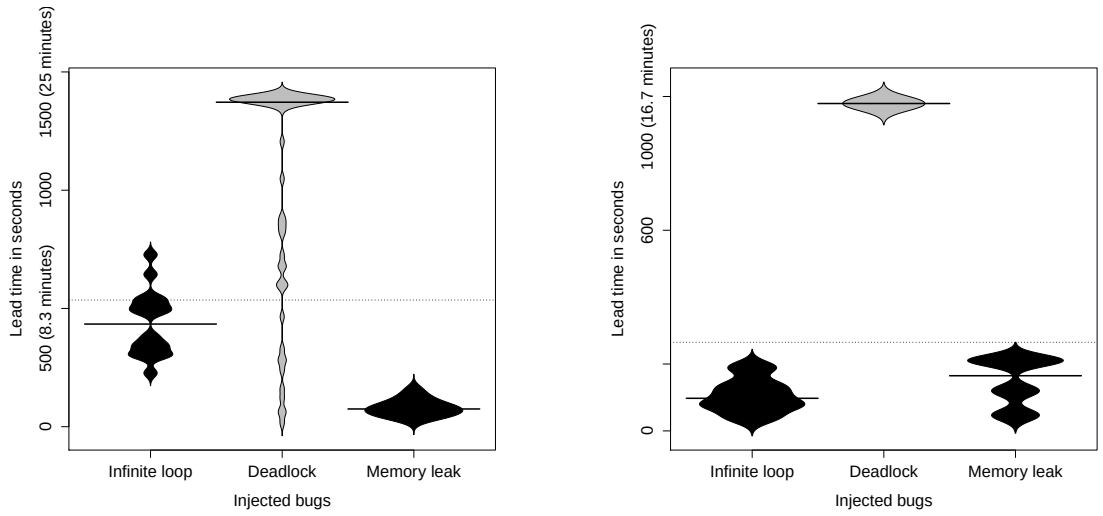
Our approach can predict performance anomalies with 97-100% precision and 80-100% recall. Our approach outperforms the UBL and the simple baselines in predicting performance anomalies in the studied applications. By analyzing the anomaly prediction examples of our approach and the baselines, we observe that our approach can handle the fluctuations in the values of the monitoring metrics using the LSTM neural networks.

3.4.2 RQ2. How early our approach can raise a performance anomaly warning before the anomaly occurs?

Motivation: To demonstrate that our approach is capable to predict performance anomalies in advance, in this RQ, we measure the lead time of our approach. Lead time calculates the amount of time that our approach can raise a performance anomaly warning before the anomaly occurs.

Approach. Dean et al. [59] mark the point when the performance anomalies occur by searching service level objective (SLO) violations (e.g., the average HTTP request response time is larger than a specific value). In particular, Dean et al. identify the performance anomalies when the average HTTP request response time is greater than 100ms. We use the same threshold (i.e., 100ms) to mark the performance anomaly occurrence in our experiment. For Elasticsearch, we mark a performance anomaly if the average response time for processing the HTTP requests is greater than 100ms. For Hadoop applications, we mark a performance anomaly when the application does not make any progress for processing the input data. As shown in Figure 3.10, if a warning is raised at the second t_p and the real performance anomaly happens at the

second $p_{p+\Delta}$, the lead time is Δ seconds. As shown in Section 3.4.1, the precision of the baselines is low, meaning that the baselines tend to raise false anomaly warnings. The lead times achieved by the baselines are not valid because many anomaly warnings are false warnings. Thus, we do not compare the lead times of our approach with the lead times of the baselines.



(a) Lead times from Elasticsearch experiments

(b) Lead times from Hadoop experiments

Figure 3.13: Beanplots of the lead times from Elasticsearch and Hadoop experiments. The dotted line represents the overall median lead time.

Results: Our approach predicts performance anomalies in advance with lead times that vary from 20 to 1,403 seconds (i.e., 23.4 minutes). Figures 3.13(a) and 3.13(b) show the lead times achieved by our approach for the studied applications. For **Elasticsearch**, the median lead time for predicting the anomalies that are caused by the deadlock bugs is 1,372 seconds (i.e., 22.8 minutes), with the maximum lead time of 1,403 seconds (i.e., 23.4 minutes). For predicting the anomalies introduced by the infinite loop bugs, our approach achieves a median lead time

of 434 seconds (i.e., 7.2 minutes). Our approach achieves a median lead time of 75 seconds for predicting the anomalies that are caused by the memory leak bugs in Elasticsearch. We believe that the short lead times for the memory leak bugs are because of the high fluctuation of the memory usage at run-time. Thus, our approach can only predict an anomaly when the application is close to the anomalous state.

For **Hadoop applications**, our approach achieves a median lead time of 165 seconds (i.e., 2.8 minutes) for predicting the performance anomalies that are caused by the memory leak bugs. For the performance anomalies that are caused by the deadlock bugs, our approach is able to predict them with a median lead time of 979 seconds (i.e., 16.3 minutes).

Compared with the anomalies caused by the infinite loop and memory leak bugs, our approach can predict the anomalies caused by the deadlock bugs with longer lead times. The deadlock bugs continuously create threads with deadlock issues, and each thread has its own memory space. With more memory being consumed by the increasing number of threads, the studied applications become slower and cause performance anomalies to happen when system run out of memory. Our approach predicts the anomalies based on the anomalous behavior of the number of threads (i.e., the number of threads continues to increase). Our approach achieves long lead times (i.e., 979 seconds) because each thread only consumes a small amount of memory, and it takes a long time to create enough threads to affect the memory usage of the studied applications.

Our approach aims to predict performance anomalies and enable operators to prevent the predicted anomalies from happening. As mentioned in the existing study [236], local anomaly prevention actions, such as scaling VM resources, can

be completed within one second and more costly anomaly prevention, such as live VM migration, takes 10 to 30 seconds to finish. The lead times achieved by our approach vary from 20 seconds to 1,403 seconds (i.e., 23.4 minutes), such lead times can be sufficient for the automatic prevention approaches [236] to take proper actions and prevent the anomalies from happening.

It should be noted that the achieved lead times can be impacted by the frequency of triggering the injected defects during the workload simulation. For example, during the workload simulation of Elasticsearch, we send the HTTP requests to trigger the injected defects. The more frequent we trigger the injected defects, the quicker the performance anomalies can occur. Thus, the shorter lead times can be achieved by our approach. In addition, the selected thresholds can impact the lead times as follows. Using a small threshold to check the differences between the predicted values of the monitoring metrics and the real observed values of the metrics can improve the lead times because a small deviation from the normal behaviors can trigger an anomaly warning. However, more false-positive predictions can be raised too when we apply a small threshold.

Summary

Our approach achieves lead times that vary from 20 to 1,403 seconds (i.e., 23.4 minutes) for predicting performance anomalies in our experiments. As suggested by the prior study [236], the achieved lead times are sufficient for automatic prevention approaches to take proper actions and prevent the predicted anomalies from happening.

3.4.3 RQ3. Could our approach be used to predict anomalies happened in the real world?

Motivation: In the previous RQs, we measure the performance of our approach in predicting the performance anomalies caused by the injected defects. However, the injected performance defects might be more straightforward than the real-world performance defects. Hence, in this RQ, we demonstrate the ability of our approach to predict the performance anomalies that are caused by real-world performance defects and compare the performance of our approach with the baselines. In addition, we evaluate the run-time overhead of our anomaly prediction approach to test whether our approach is practical to predict anomalies in real-world systems.

Approach: To obtain performance defect issue reports, we search issue reports that contain keywords (e.g., memory leak, deadlock, and infinite loop) about performance defects in the studied applications. No performance defect issue reports are returned after we search for the Hadoop MapReduce applications. The MapReduce applications are small (i.e., less than 1,000 lines of code) and are used as sample applications for learning purposes. Thus, it is reasonable that no performance defects have been found in these sample applications. Therefore, we could not reproduce performance defects in MapReduce applications.

For each identified performance defect issue report, we obtain the Elasticsearch version that contains the performance defect. Next, we identify the operations that can trigger the performance defect by exploring the issue reports and analyzing the source code of the buggy version. There are performance defects that only occur under special workloads and running environments. For example, the infinite loop bug

#30962[3] only happens in Windows platforms and the deadlock bug #36195[4] requires Elasticsearch to run on a multiple-node cluster. Due to our single-node experiment environment, we could not reproduce performance defects that need intensive datasets or require the tested application to run on large clusters (e.g., deploying Elasticsearch to several virtual machines). Because of the limited and ambiguous information in issue reports, reproducing real-world performance defects is a challenging and time-consuming task [216]. To this end, we select and reproduce five performance defects that can be reproduced in our single-node experiment environment and can be triggered automatically through scripts. The reproduced bugs are as listed in Table 3.9. We reproduce more memory leak bugs than the infinite loop and deadlock bugs because memory leak bugs usually do not require a multiple-node cluster environment to reproduce.

To collect the values of the monitoring metrics in anomalous executions, we follow the same approach shown in Figure 3.5. We send HTTP requests that trigger the real-world performance defects while running the macrobenchmarking test suites. Each Elasticsearch version has its supported test suites. In total, we apply 16 test suites to the five buggy Elasticsearch versions shown in Table 3.9. To collect more anomalous executions, we iterate the workload simulation five times [26]. To this end, we collect 16*5 (i.e., 80) anomalous Elasticsearch executions with performance anomalies. For the data collected from each workload simulation iteration (i.e., 16 anomalous Elasticsearch executions), we evaluate the precision, recall, and F-score of our approach

[3] https://github.com/elastic/elasticsearch/issues/30962
[4] https://github.com/elastic/elasticsearch/issues/36195
[5] https://github.com/elastic/elasticsearch/issues/6553
[6] https://github.com/elastic/elasticsearch/issues/8249
[7] https://github.com/elastic/elasticsearch/issues/9377
[8] https://github.com/elastic/elasticsearch/issues/24108
[9] https://github.com/elastic/elasticsearch/issues/24735

Table 3.9: Summary of the reproduced real-world performance defects.

Bug ID	Description	Performance defect type	Elasticsearch version
#6553[5]	Out of memory issue occurs when using percolator queries.	Memory leak	1.2.1
#8249[6]	Heap usage grows when using cache keys.	Memory leak	1.3.4
#9377[7]	Elasticsearch hangs when using the collate option.	Deadlock	1.4.2
#24108[8]	High memory usage issue occurs when using nested queries.	Memory leak	5.5.0
#24735[9]	Thread falls into infinite loop when processing indices queries.	Infinite loop	5.3.1

and the baselines in predicting anomalies. We use the same prediction models and thresholds that are trained and validated on the datasets shown in Table 3.4. After five iterations, we calculate the average and the standard deviation of the precision, recall, and F-score of our approach and baselines in predicting performance anomalies. To evaluate the run-time overhead of our anomaly prediction approach, we measure the amount of time, CPU usage, and memory usage that our approach takes to make anomaly predictions. We conduct the experiments on our experiment environment with an 8-core Intel i7-4790 3.60GHz CPU and 32GB memory.

Results: Our approach achieves high precision and recall in predicting the performance anomalies that are caused by the five reproduced performance defects in Elasticsearch. As shown in Tables 3.10 and 3.11, our

approach achieves an average of 100% precision and 87% recall in predicting performance anomalies that are caused by the deadlock bugs. In addition, our approach achieves average 100% and 95% precision in predicting performance anomalies caused by the infinite loop and memory leak bugs.

Table 3.10: The precision of our approach and the baselines in predicting performance anomalies that are caused by the real-world performance defects. We highlight the results of the approaches that achieve the best performance.

	Deadlock	Infinite loop	Memory leak
Our approach	**100%±0%**	**100%±0%**	**95%±9%**
The UBL baseline	49%±3%	59%±4%	71%±15%
The simple baseline	50%±0%	83%±24%	68%±19%

Table 3.11: The recall of our approach and the baselines.

	Deadlock	Infinite loop	Memory leak
Our approach	87%±9%	**100%±0%**	**100%±0%**
The UBL baseline	**100%±0%**	**100%±0%**	**100%±0%**
The simple baseline	**100%±0%**	**100%±0%**	**100%±0%**

Table 3.12: The F-score of our approach and the baselines.

	Deadlock	Infinite loop	Memory leak
Our approach	**93%±5%**	**100%±0%**	**97%±5%**
The UBL baseline	66%±2%	74%±3%	82%±11%
The simple baseline	67%±0%	89%±16%	79%±13%

Our approach achieves higher average performance with less deviation compared with the baselines. As shown in Table 3.12, our approach consistently achieves higher precision, recall, and F-score in predicting performance anomalies that are caused by the different real-world performance defects than the baselines. In addition, our approach achieves less deviation. Less deviation means that our approach achieves more stable performance in predicting performance anomalies than

the baselines. For example, the deviation of the F-score of our approach in predicting anomalies that are caused by memory leak bugs is 0.05, while the deviation of the UBL approach is 0.11.

Our approach predicts the performance anomalies caused by the real-world performance defects with lead times that vary from 2 to 846 seconds (i.e., 14.1 minutes). As shown in Figure 3.14, the median lead time for predicting the anomalies that are caused by the infinite loop bug is 116 seconds (i.e., 1.9 minutes). For predicting the anomalies that are caused by the memory leak bugs, our approach achieves a median lead time of 406 seconds (i.e., 6.7 minutes), with the maximum lead time of 846 seconds (i.e., 14.1 minutes). Our approach achieves the median lead time of 3 seconds for predicting the anomalies that are caused by the deadlock bug.

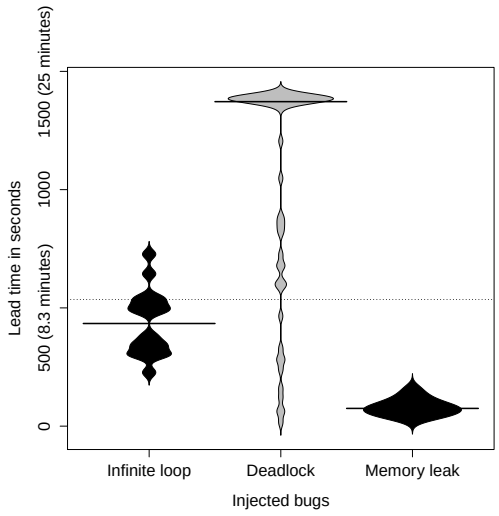


Figure 3.14: Beanplots of the lead times for predicting the performance anomalies caused by the five reproduced performance defects in Elasticsearch. The dotted line represents the overall median lead time.

We analyze the reason behind the short lead times for predicting the anomalies caused by the deadlock bug #9377. The deadlock bug #9377 causes the Elasticsearch

to hang with the searching threads deadlocking each other. The searching threads in Elasticsearch are responsible for processing document searching requests. After all the searching threads deadlock each other, Elasticsearch stops processing document searching requests and puts searching requests into the waiting queue. After the waiting queue becomes full, Elasticsearch starts throwing out exceptions and rejecting more searching requests. Because Elasticsearch is used to search documents in practice, during the workload simulation, intensive document searching requests (e.g., hundreds in a second) are sent to Elasticsearch. In our experiments, we use the default queue size of 1,000 [74]. After the deadlock bug occurs, the waiting queue of Elasticsearch is filled with a thousand of searching requests within a few seconds, and Elasticsearch starts rejecting additional searching requests. The 100% precision shown in Table 3.10 means that our approach is able to predict the performance anomalies caused by the deadlock bug #9377 when the anomalies happen in a few seconds after the occurrence of the performance defect.

We manually analyze the *false positive* predictions that are raised by our approach. Figure 3.15 shows a false positive prediction example when we test our approach using the data collected from the memory leak bug #24108. We run Elasticsearch under a workload that requires Elasticsearch to load a large number of documents into the memory. f Thus, the real observed memory usage fluctuates from the beginning, and it is difficult to predict the fluctuating value precisely. As shown in Figure 3.15 (c), our trained LSTM neural network can predict fluctuating memory usage with small differences. After a memory leak bug starts to occur at the time mark of 1,163 second, the real observed memory usage continues to increase and reaches to 100% usage of the memory space. A performance anomaly is observed at the time mark of 1,968

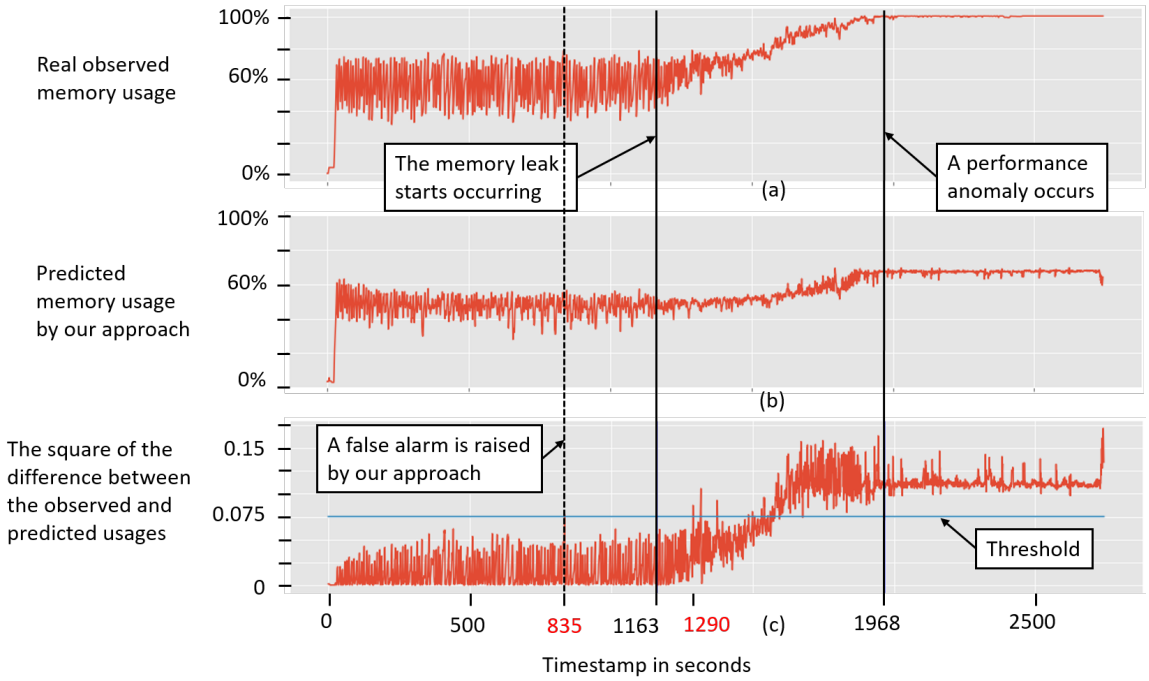


Figure 3.15: A false positive prediction that is raised by our approach. Timestamps when our approach raises anomaly warnings are highlighted in red.

second based on the increase in the average response times of the HTTP requests (as described in Section 3.3.1). As shown in Figure 3.15 (b), our approach predicts that memory usage should stop growing at around 70%. The differences between the observed and predicted memory usages start increasing before the occurrence of the anomaly. At the time mark of 1,290 second, the difference exceeds the threshold (i.e., 0.075), and our approach raises a true anomaly warning. As introduced in Section 3.4.1, the threshold (i.e., 0.075) is selected by maximizing the F-score on the Elasticsearch validation dataset shown in Table 3.4. However, before the memory leak bug starts to occur, the difference between the observed and the predicted memory usages exceeds the threshold (i.e., 0.075) at the time mark of 835 second, and our approach raises an anomaly warning. There is no performance defect occurring at

the time mark of 835 second and Elasticsearch is under normal executions. Hence, the anomaly warning is false, and our approach makes a false positive prediction.

Table 3.13: The measurements of the overhead of our approach

Number of samples	Time taken (in seconds)	Average CPU usage	Maximum CPU usage	Average memory usage	Maximum memory usage
141,647	36	21.8%	51.6%	2.0%	2.1%

The run-time overhead of our performance anomaly prediction approach is feasible for real-world systems. As shown in Table 3.13, there are 141,647 samples in the monitoring data collected from the replicated performance defects. For each sample (i.e., the values of the monitoring metrics collected at each second), our approach processes the collected values of the monitoring metrics and predicts if a performance anomaly would happen. Our approach takes 36 seconds to process the 141,647 samples and predict performance anomalies that are caused by the real-world performance defects. On average, our approach needs 36/141,617 seconds (i.e., 0.25 milliseconds) to predict anomalies using the values of the monitoring metrics collected at every second. The average CPU usage of our approach for predicting anomalies is 21.8% on our machine. There are only 8 cores in our CPU, which means that our approach needs a average of 8*21.8% (i.e., 2) CPU cores to predict anomalies. The average memory usage of our approach is 32GB*2.0% (i.e., 320MB). More powerful CPUs and larger memory size than our experiment machine are used in the industrial environments. For example, in Amazons Elastic Compute Cloud (EC2), each node (i.e, virtual machine) can have up to 16 cores of 2.3 GHz AWS Graviton CPU and 32GB memory [7]. In the industrial environments, our approach can be assigned to separate CPU cores from the monitored application and do not

impact the executions of the monitored application. Thus, our anomaly prediction approach is practical to predict anomalies in real-world systems.

Summary

Our approach achieves higher average performance with less deviation compared with the baselines in predicting the performance anomalies caused by the five real-world performance defects. Our approach achieves lead times that vary from 2 to 846 seconds (i.e., 14.1 minutes). In addition, our approach takes an average of 0.25 milliseconds, 320MB memory, and 25% CPU usage to predict anomalies at every second.

3.5 Discussion

In this section, we discuss the implementation of our approach to monitor applications in the real-world industrial environment, the usage scenarios of our approach, and the limitations of our approach.

3.5.1 Implementing our approach to monitor applications in the industrial environment

Data collection. Figure 3.16 shows the overview of implementing our approach to monitor an application. As shown in Figure 3.16, the first step to implement our anomaly prediction approach is to collect data to train the LSTM neural networks. For an application, the perfect training data should be collected from monitoring the latest stable version of the application. Human operators can manually examine the collected data to make sure that no performance anomaly happened during the data collection.

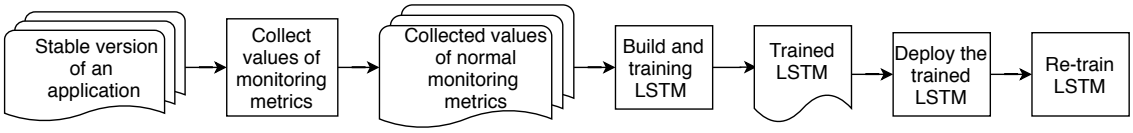


Figure 3.16: The overview of implementing our approach to monitor an application.

Building and training LSTM neural networks. After data collection, the parameters for building and training the LSTM neural networks should be adjusted based on the collected data as shown in Figure 3.16. For example, the number of input features should be the same as the number of monitoring metrics in the collected data. Once the parameters are decided, the LSTM neural networks can be trained offline using the collected data.

Deploying and re-training LSTM neural networks. The trained LSTM neural networks can be deployed on the platform (e.g., VMs in a cloud infrastructure) which hosts the application to be monitored. After the deployment, the LSTM neural networks can track the values of the monitoring metrics of the application as input and predict performance anomalies. In addition, the behaviors of the application might change because of the application updating. To adapt the LSTM neural networks to the new behaviors, the LSTM neural networks should be re-trained using the data collected from the new stable versions of the application as shown in Figure 3.16. However, frequently re-training can decrease the performance of the LSTM neural networks if the neural networks are trained on a non-stable version, i.e., a version that has buggy behaviors but are not realized by developers.

3.5.2 The usage scenarios of our approach

We deploy our approach to monitor the application-level metrics. In this section, we describe two scenarios of deploying our approach.

Scenarios when an application runs on a multiple-node cluster. Applications can be hosted on a cluster with thousands of nodes (e.g., virtual machines). For monitoring an application running on a large-scale cluster, we can first collect application-level metrics of the instance of the application (i.e., the application process) running on each node (e.g., virtual machine) in the cluster. Then, we can train the LSTM neural network using the collected application-level metrics values. Next, our approach can be deployed into each node in the cluster. For every node, our approach monitors the instance of the application running on the node and predicts anomalies for the instance of the application. By predicting anomalies for each instance of the application, our approach can help developers and operators to locate the anomalous instance of the application in the cluster.

Scenarios when several applications are running in the same system. In an industrial environment, there might be several applications running on the same machine or virtual machine (VM). The co-existing applications share computation resources, such as CPU and memory. The anomalous behaviors of one application can affect the executions of other co-existing applications. For example, a memory leak bug in an application can cause the system running out of memory and affect the executions of other applications in the system. For the scenarios when several applications are running in the same system, operators can deploy multiple instances of our approach. More specifically, each instance monitors an application and predicts the performance anomalies for the application. By monitoring each application, our approach can predict which application is the anomalous application. For example, applications A and B are running in the same system and there is a memory leak

happening in the application B. After deployment of separate instances of our approach to each application, our approach can predict performance anomalies in the application B based on its increasing memory usage.

3.5.3 The limitations of our approach

Our approach captures the normal behaviors of applications using the LSTM neural networks. In our experiments, we observe that our approach achieves good performance in predicting performance anomalies by checking whether applications deviate from the captured normal behaviors. However, every approach has its limitations. In this section, we summarize and discuss the limitations of our approach.

Our approach may not be able to differentiate non performance anomalies that deviate from normal behaviors from performance anomalies. For example, a spontaneous increase in the number of documents loading requests to Elasticsearch can suddenly increase the CPU and memory usages of Elasticsearch . In our experiments, we observe that our approach can handle the fluctuations in the monitoring metrics (e.g., CPU and memory usages). However, if a workload change results in large fluctuations in the monitoring metrics, our approach might raise a false performance anomaly warning. For example, our approach makes a false positive prediction when the memory usage is fluctuating heavily as shown in the example in Section 3.4.3.

Our approach has limitations to automatically adapt to the new normal behaviors of applications. The LSTM neural networks capture only the normal behaviors that are included in the training dataset. However, the behaviors of applications could change over time because of the updates of the application. In our

approach, the LSTM neural networks cannot automatically adapt to the new behaviors of applications at the run-time monitoring phase. The LSTM neural networks are required to be re-trained offline to learn the new behaviors of applications.

3.6 Threats to Validity

In this section, we discuss the threats to the validity of experiments conducted in this chapter.

Threats to external validity is related to the generalizability of our results with respect to other project settings. To ensure the generalizability of our approach, we conduct experiments on two JAVA software systems. These two software systems have different architectures and belong to different domains. However, studying software systems that are programmed in other languages can be useful to augment the generalizability of our approach.

Threats to internal validity concern the uncontrolled factors that may affect the experiment results. One internal threat to our results is that the injected performance defects are simpler than real-world performance defects. To mitigate this threat, we test our approach on real-world performance defects by reproducing the performance defects of Elasticsearch. We manage to reproduce five real-world performance defects of Elasticsearch. The obtained results show that our approach can accurately predict the performance anomalies that are caused by real-world performance defects. However, further studies can explore reproducing more performance defects, such as concurrent performance defects. The time interval that is used to collect the values of the monitoring metrics of the studied systems can affect our results. Different time intervals might affect the ability of our approach for monitoring

software systems and predicting performance anomalies. We apply one second time interval following the existing work [59]. However, further studies can explore to set the best time intervals for different software systems. In addition, the thresholds that are used to raise performance anomaly warnings can affect our results. There is a tradeoff between precision and recall. Using a smaller threshold can improve the recall of our approach because small deviations from the normal behaviors can raise performance anomaly predictions. However, a smaller threshold can affect the precision of our approach since the predictions raised from small deviations might be false-positive predictions. In our experiments, we determine the values of the thresholds by selecting the values that achieve the best F-score on the validation datasets.

3.7 Summary

In this chapter, we present an approach that predicts performance anomalies in software systems. First, our approach uses LSTM neural networks to capture the normal behaviors of software systems. Then, our approach predicts performance anomalies by analyzing run-time monitoring data and checking the early deviations from the normal behaviors that are expected from the LSTM neural networks. We conduct extensive experiments using two real-world software systems (i.e., Elasticsearch and Hadoop applications) to evaluate our approach. In addition, we demonstrate the ability of our approach to predict the performance anomalies that are caused by real-world performance defects.

The obtained results show:

- Our approach outperforms the baselines and predicts various performance anomalies with 97-100% precision and 80-100% recall.

- Our approach predicts performance anomalies in advance with lead times that vary from 20 to 1,403 seconds (i.e., 23.4 minutes).
- Our approach achieves 95-100% precision and 87-100% recall with lead times that vary from 2 to 846 seconds (i.e., 14.1 minutes) for predicting the anomalies that are caused by the five real-world performance defects in Elasticsearch.

Chapter 4

Enhancing Performance Defect Prediction Using Performance Code Metrics

Performance defects are non-functional defects that can significantly reduce the performance of an application (e.g., software hanging or freezing) and lead to poor user experience. Prior studies found that each type of performance defects follows a unique code-based *performance anti-pattern* and proposed different approaches to detect such anti-patterns by analyzing the source code of a program. However, each approach can only recognize one performance anti-pattern. Different approaches need to be applied separately to identify different performance anti-patterns. To predict a large variety of performance defect types using a unified approach, we propose an approach that predicts performance defects by leveraging various historical data (e.g., source code and code change history).

Chapter organization: In Section 4.1, we introduce this chapter. Section 4.2 presents the preliminary study of analyzing the characteristics of performance defects. Section 4.3 describes the experimental setup, while Section 4.4 presents our results. We discuss the usages and limitations of our approach in Section 4.5. The threats

to validity are discussed in Section 4.6. We summarize the chapter and discuss the future work in Section 4.7.

4.1 Introduction

Large-scale software systems are becoming increasingly more prominent in our society. High performance is critical for the user perception and the quality of software systems. However, user perception and software system's quality can be affected negatively by performance defects such as software hanging or freezing [193]. For instance, performance defects (e.g., memory leak bugs) can deteriorate the responsiveness and throughput of software systems, which results in poor user satisfaction and waste of computational resources [185, 192]. Such bugs exist widely in released software systems, even in well tested commercial products such as Windows 7s Windows Explorer [106, 184].

Prior studies [46, 171, 192, 275, 276] study the characteristics of performance defects and find that fixing performance defects is more time-consuming compared to non-performance defects. Similarly, prior study [286] finds that most performance issues are caused by poor architectural decisions and performance defects fixes usually require design-level optimizations instead of simple code changes. Thus, it is important to predict performance defects to provide developers warnings at an early stage of software development phase and help developers fix performance defects (e.g., conduct design-level optimization [286]).

Various prior approaches [53, 86, 191, 195, 280] have been proposed to recognize performance defects at early stage of the development phase. For example, anti-patterns are design and implementation styles which lead to poor source code

quality [232] and existing studies identify anti-patterns that lead to performance defects, such as data corruption hang bugs [53], redundant traversal bugs [195], memory and resource leak bugs [86], and synchronization bugs [280]. The anti-patterns (e.g., the exit condition of a loop depends on I/O operations) are used as restricted rules to check if source code contains performance defects. In addition, prior defect prediction studies [19, 22, 31, 64, 108, 161, 181, 281, 283, 284] use code and process metrics that are derived from source code and code change history to predict defects (not specific to performance defects). However, the prior approaches have the following limitations.

- **Limitation of prior defect prediction approaches**. The code and process metrics used in defect prediction approaches are designed to detect any type of defects. Therefore, the prior defect prediction approaches [19, 22, 31, 64, 108, 161, 181, 281, 283, 284] do not consider the code characteristics specific to performance defects, which may impact the accuracy of predicting performance defects.

- **Limitations of prior anti-pattern based approaches**. Each approach can only identify one performance anti-pattern. It is time-consuming to configure and apply different approaches separately to identify different performance anti-patterns. Moreover, prior approaches [53, 86, 191, 195, 280] cannot predict the performance defects that do not follow the identified anti-patterns.

To address the limitations in prior approaches, we propose a set of performance code metrics that capture the code characteristics that can lead to poor performance.

Our work combines performance code metrics with the source code and process metrics used in defect prediction studies to build models for predicting performance defects at file level. Different from the prior studies [53, 86, 191, 195, 280], the performance code metrics we propose measure code features (e.g., the number of I/O operations and the number of loops) instead of finding restricted rules that match anti-patterns. Our approach can predict various types of performance defects. To evaluate our approach, we conduct experiments on 80 open-source Java projects obtained from GitHub. Our work aims to address the following research questions (RQs):

RQ1. What is the performance of our approach in predicting performance defects at file level?

We utilize seven well-known machine learning algorithms used in prior defect prediction studies [5, 238] such as Random Forest, Naive Bayes, and Logistic Regression. Our findings suggest that Random Forest and eXtreme Gradient Boosting machine learning algorithms achieve the best performance with a median value of 0.84 AUC, 0.21 Precision-Recall AUC (PR-AUC), and 0.38 Matthews Correlation Coefficient (MCC) for predicting performance defects at file level.

RQ2. Which group of metrics affect the performance of our models the most?

In this RQ, we study the performance effects of the group metrics on seven machine learning models. To build machine learning models, we use three groups of metrics (i.e., code metrics, process metrics, and performance code metrics). We observe that the proposed performance code metrics are the most important metrics in the studied machine learning models. Specifically, the AUC, PR-AUC, and MCC of the seven

studied machine learning models drop a median of 7.7%, 25.4%, and 20.2% without using the proposed performance code metrics.

RQ3. What are the different types of performance defects that our approach can predict?

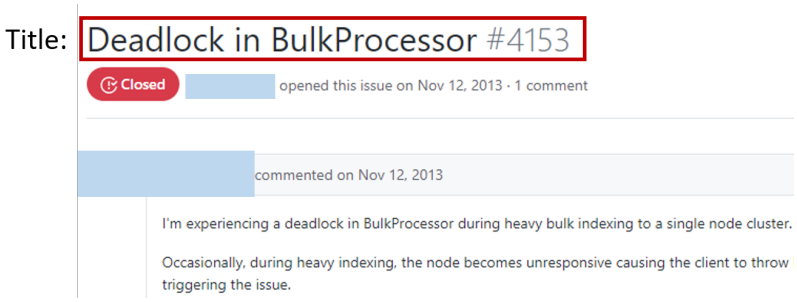
In this RQ, we study the types of performance defects that our approach can predict. We find that our approach can predict various types of performance defects, including performance regression bugs, memory leak bugs, infinite loop bugs, deadlock bugs, and hang bugs. In addition, our approach can predict additional performance defects that are not covered by the anti-patterns proposed in the prior studies [53, 86, 191, 195, 280].

4.2 Preliminary Study. What are the characteristics of performance defects in comparison to non-performance defects?

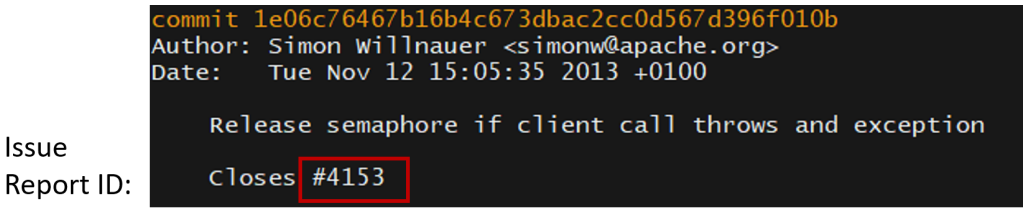
Motivation: Before predicting performance defects in software systems, we intend to have a better understanding on them, e.g., do performance defects take longer time to fix than non-performance defects? are performance defects more time-consuming to fix than non-performance defects? Various prior studies [46, 171, 192, 275, 276] have found that performance defects take longer time and more developers to fix than non-performance defects. However, the prior studies [46, 171, 192, 275, 276] analyze performance defects and non-performance defects from only two to five projects (e.g., Mozilla Firefox and Google Chrome). It is important to confirm whether the findings of the prior studies (e.g., performance defects take longer time to fix) hold on a larger dataset of projects.

Downloading issue reports. As a first step, we collect and classify issue reports

using the GitHub API [90]. We collect only the closed issue reports of 80 experiment GitHub projects (see the project selection criteria in Section 4.3).



(a) A snapshot of the issue report of a deadlock performance defect in Elasticsearch project. The identify of the developer is covered for the purpose of privacy.



(b) The fixing commit for the reported deadlock performance defect

Figure 4.1: An example of a performance defect report and its fixing commit.

Classifying issue reports. We classify the collected reports into performance and non-performance defect reports. To identify the performance defect issue reports, we apply a search query—comprised by various performance-related keywords—on title fields of issue reports similar to the prior studies [46, 52, 56, 57, 276]. Specifically, we use the following performance-related keywords: *deadlock*, *contention*, *infinite loop*, *memory leak*, *performance*, *high memory*, *stuck*, *hang*, *slow*, *speed up*, and *100% CPU*. If a issue report matches any of the above keywords, we label it as a performance defect issue report, otherwise as a non-performance defect issue report. Figures 4.1(a) and 4.1(b) show a performance defect issue report and the commit that fixes the reported performance defect. Figure 4.1(a) illustrates an identified performance defect

issue report with the *deadlock* keyword in its title field. The issue report is submitted by developers to report a *deadlock* bug in the Elasticsearch project. Figure 4.1(b) shows the commit that fixes the reported performance defect issue report. As shown in Figure 4.1(b), the message of the commit contains the issue report ID #4153

In total, we collect 2,920 performance defect issue reports. To check whether there are false positive performance defect issue reports (i.e., non-performance defect issue reports that are misclassified as performance defect issue reports), we conduct a manual analysis. We randomly select a sample of 340 performance defect issue reports using 95% confidence level and 5% confidence interval. We analyze each issue report and check whether a issue report is a performance defect issue report.

Table 4.1: The patterns of the false positive performance defect issue reports

Category of issue reports	**Patterns of issue reports**	**Number of issue reports**
issue reports about monitoring performance data	*performance matrix*, *performance data*	9
issue reports about performance test	*performance test*	7
issue reports about functional bugs	*stuck*	7
But reports about testing of slow mode operations	test*slow	7
Documentation	*DOCS*, *documentation*	2

Through the manual labelling on the statistical sample, we identify 40 (i.e., 40/340 = 11.8%) false positive performance defect issue reports. To reduce the false positives, we manually check the false positive performance defect issue reports from the

sampled dataset and summarize their patterns. These patterns are depicted in Table 4.1. Then, we apply regular expressions to remove the reports that match our patterns from entire performance defect issue reports dataset. In total, through applying regular expressions, we identify 167 false positive performance defect in our entire dataset of performance defect issue reports. After excluding the false positives, there are 2,753 (i.e., 2920-167) performance defect issue reports. To check whether there are still many false positive performance defect issue reports left in the dataset, we conduct a manual analysis of the remaining performance defect issue reports. We randomly select a second statistical sample of 340 performance defect issue reports using 95% confidence level and 5% confidence interval from the remaining 2,753 performance defect issue reports and manually label the sampled dataset again. Then, we use Cohens Kappa and obtain the score of 0.83 (i.e., almost perfect agreement between two co-authors [179]) between the labelling of the first two co-authors. Through the manual analysis, we find only 12 (i.e., $12/340 = 3.5\%$) false positive performance defect issue reports from the selected sample. As shown in Figure 4.2, our projects have a median number of 17 performance and 1,972 non-performance defect issue reports.

Analyzing the collected issue reports. We measure the characteristics of the collected performance defects and non-performance defects in terms of (1) the time taken to assign a bug to a developer, (2) the time taken to fix a bug, (3) the number of developers participating in fixing a bug, and (4) the number of developer comments when discussing how to solve a bug. The time taken to assign a bug is the amount of time needed from the moment a bug is reported until the time it is assigned to a developer. The time taken to fix a bug is the amount of time from the moment a bug

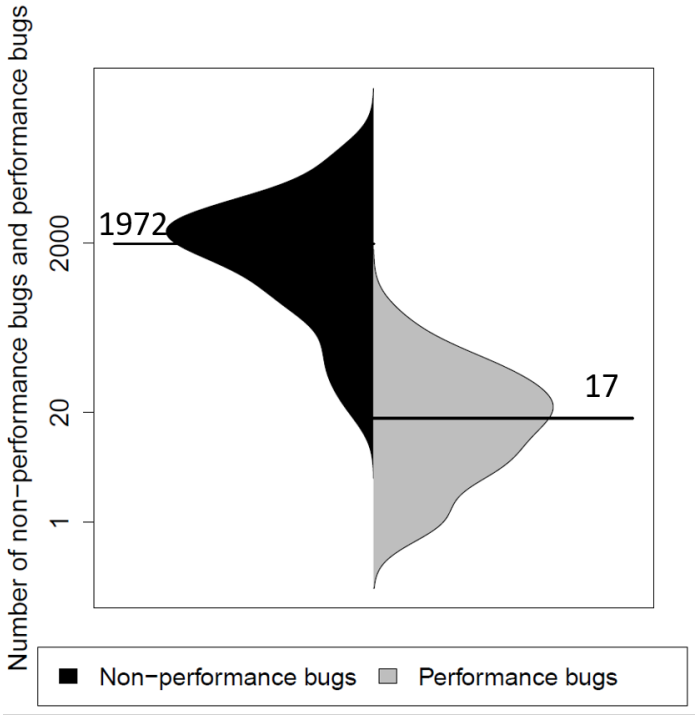
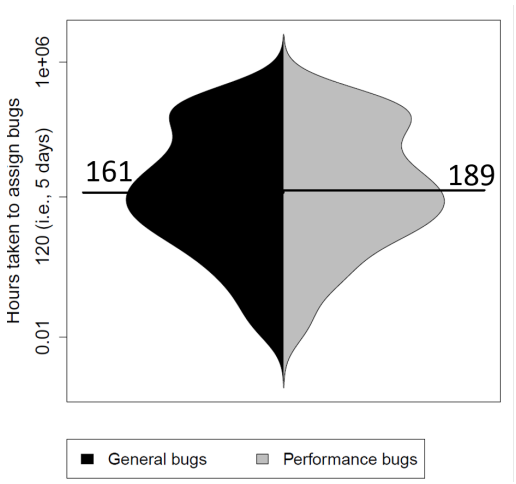


Figure 4.2: The numbers of performance defect and non-performance defect issue reports in each project.

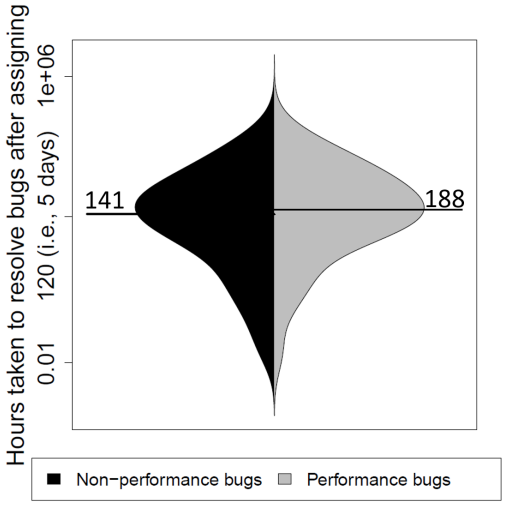
is assigned to a developer until the time it is marked as fixed.

As depicted in Figure 4.3, we visualize the characteristics of the performance defects and non-performance defects using beanplots [145]. We apply the Wilcoxon Rank Sum test [175] to statistically test if the characteristics of the performance defects and non-performance defects are significantly different. The Wilcoxon Rank Sum test does not need any parameters or assumptions about the distributions of the characteristics. We use 0.01 as the significance level when conducting the Wilcoxon Rank Sum tests. In addition, we use the Cliff's delta to quantify the magnitude of the differences. The Cliff's delta values range from -1 to +1, where a zero delta value signifies that two distributions have the same magnitude of values. As suggested by the prior studies [214], we map the Cliff's delta values (i.e., d) to significance levels as

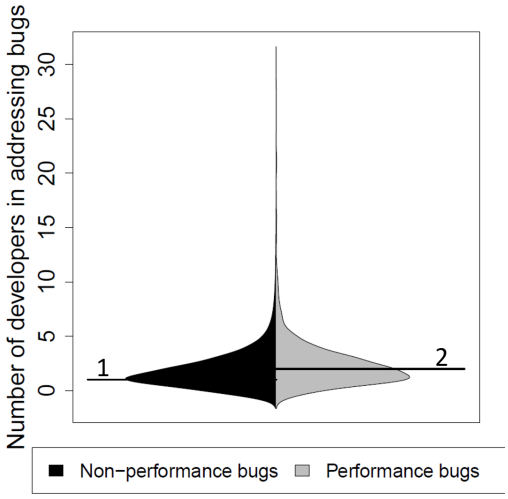
follows. $Negligible : 0 \leq |d| < 0.147; Small : 0.147 \leq |d| < 0.330; Medium : 0.330 \leq |d| < 0.474; and Large : 0.474 \leq |d| < 1$



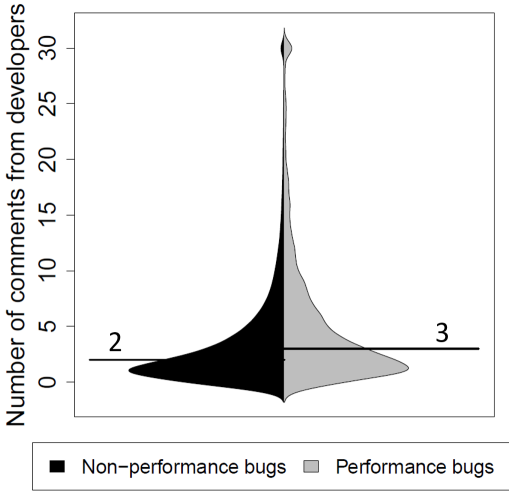
(a) The time taken to assign performance defects and non-performance defects to appropriate developers



(b) The time taken to fix performance defects and non-performance defects after assigning them to appropriate developers



(c) The number of developers participating in fixing bugs



(d) The number of comments from developers in fixing bugs

Figure 4.3: The beanplots of the characteristics of the performance defects and non-performance defects.

Table 4.2: The results of the Wilcoxon Rank Sum tests

Comparison	**P-value of the Wilcoxon Rank Sum test**	**Cliff's Delta effect size**
The **time taken to assign** performance defects and non-performance defects	0.18	0.03 (negligible)
The **time taken to fix** performance defects and non-performance defects	3.04e-07	0.11 (negligible)
The **number of developers** in fixing performance defects and non-performance defects	1.14e-13	0.08 (negligible)
The **number of comments** in fixing performance defects and non-performance defects	2.20e-16	0.18 (small)

Results: It is more time-consuming to solve performance defects than non-performance defects. The characteristics of the performance defects and non-performance defects are shown in Figures 4.3(a), 4.3(b), 4.3(c), and 4.3(d). First, the performance defects take 32.3% (i.e., $\frac{188-142}{142}$) more time to solve than the non-performance defects as shown in Figure 4.3(b). The median time of fixing the non-performance and performance defects are 142 and 188 hours, respectively. As shown in Table 4.2, the $p-value$ of the Wilcoxon Rank Sum test of the time taken to solve the performance defects and non-performance defects is less than 0.01. The time taken to solve the performance defects is significantly different from the time taken to solve non-performance defects. In addition, more developers participate in fixing the performance defects than non-performance defects as shown in Figure 4.3(c). The median number of developers participate in fixing the non-performance and performance defects are one and two, respectively. Nevertheless, as shown in Figure 4.3(d),

the performance defects receive more comments than the non-performance defects, which means that developers discuss more when fixing a performance defect. The $p - values$ of the Wilcoxon Rank Sum tests are less than 0.01 for the comparisons of the number of developers and the number of comments. Thus, the number of developers and the number of comments are significantly different between the performance defects and the non-performance defects. As shown in the Table 4.2, the effect size is 0.18 (i.e., small difference magnitude categorized by the Cliff's Delta test) between the number of comments from developers in fixing performance defects and non-performance defects, while the effect sizes between other groups of comparisons are significantly although negligible.

Summary

We confirm that performance defects are more troublesome to fix than non-performance defects on a large scale of experiment projects. Thus, it is important to predict performance defects to provide developers warnings at an early stage of the development phase and help developers fix performance defects (e.g., conduct design-level optimization [286]) before committing the code to the repositories.

4.3 Experiment Setup

In this section, we introduce the overview of our experimental setup. As shown in Figure 4.4, we first collect data from GitHub and identify performance defects. Then, we calculate various types of metrics to capture the characteristics of source code and build performance defect prediction models.

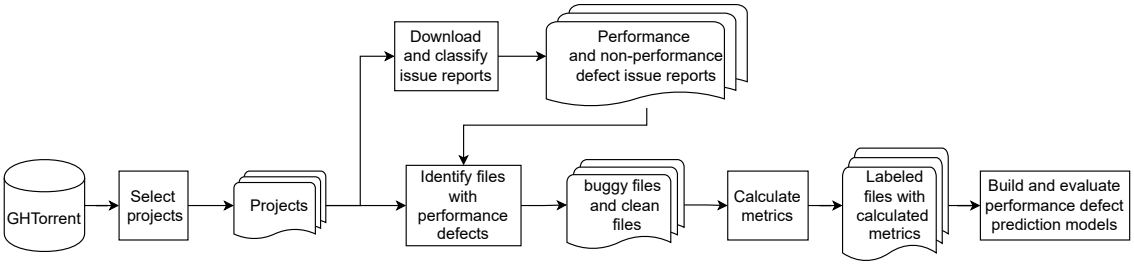


Figure 4.4: The overview of our approach.

4.3.1 Selecting projects

As shown in Figure 4.4, we first query the project information hosted on GHTorrent [94] to select our projects. Then, we clone the selected projects from GitHub. GHTorrent is a widely used dataset that hosts data collected from GitHub [95, 97, 274]. We defined the following criteria to select our projects.

Selection Criteria. As Java is one of the most popular programming languages according to programming language popularity websites (e.g., Tiobe [243], GitHut [91], and PYPL [210]), we restrict our analysis only to Java projects. Nevertheless, we believe that our approach can be adapted to other programming languages, assuming that the metrics are properly calculated. To avoid working on personal or toy projects [143], we select 1,697 Java projects that each of them has at least 2,000 commits [221]. Prior studies [282] found that many open-source projects do not have long development history or history bugs for building and evaluating bug prediction models. Thus, we apply the following criteria to further exclude projects from the initial selection:

- In the work by Gousios et al. [93], more than half of the GitHub projects are forked from other repositories. To avoid studying inactive or personal projects, we exclude projects that have been archived [89] or forked.

- We exclude projects that do not have issue reports or are managed by issue tracking systems (e.g., Jira and Bugzilla) [143]. We limit our analyses to the projects that have issue reports on GitHub as we intend to use issue reports to identify performance defects in source code. Studying GitHub issue reports alone allows us to implement scripts to automatically parse issue reports. Nevertheless, our approach can be adapted to projects that use external issue tracking systems, assuming that the issue reports can be automatically downloaded and parsed.

- We filter out projects with a lifespan less than a year, otherwise we can not have enough performance defect reports and metrics to build accurate prediction models [281, 282, 289].

- We exclude projects that have limited performance defect fixing commits. We need enough performance defects in a project to build and evaluate performance defect prediction models. performance defect fixing commits are used to identify performance defects in a project. Following the prior studies [282], we count the number of performance defect fixing commits from a year period in each project and choose the 75% percentile of the number of performance defect fixing commits (i.e., 74 performance defect fixing commits) as the threshold to filter out projects.

- We remove projects that do not have performance defect fixing commits in the last six months periods. The last six months period means the six months by looking back from the latest commit in a project. Following the prior studies [283], we choose the six months period and we consider projects that have no

performance defect fixing commits from the last six months period as abnormal projects. Therefore, we remove such projects from our dataset.

After applying the aforementioned criteria, we reduce our projects to 80. Our projects are from different fields, such as Elasticsearch search engine [75] OpenLiberty cloud micro-service [125], and so on. On average, the collected projects have 304,619 source code files and 34,268,345 lines of code.

4.3.2 Process of manual analyses

In our experiment, we conduct several manual analyses to validate the collected performance defect reports, study the reasons why performance defects are time-consuming to fix, and check the types of performance defects that our approach can predict and the performance defects that our approach fails to predict. The process of manual analysis consists of three steps:

- We randomly select a number of sample data from an entire dataset using a 95% confidence level and 5% confidence interval to ensure that the sample size is statistically large [127].
- To mitigate the error of manual analysis, two co-authors independently analyze each sample data (e.g., a performance defect report) and record their finding about the data (e.g., whether the performance defect report is valid) .
- We compare the results from the two co-authors and use Cohens Kappa to measure the agreement between the results from the two co-authors. Then, the two co-authors discuss the sample data where they have different opinions until reaching a consensus.

4.3.3 Identifying performance defect fixing commits

We use two methods to identify the commits that fix performance defects.

- The first method is to point out the performance defect fixing commits by searching for commits containing any of the performance-related keywords in the commit message (i.e., *deadlock*, *contention*, *infinite loop*, *memory leak*, *performance*, *high memory*, *stuck*, *hang*, *slow*, *speed up*, and *100% CPU*).

- The second method is to identify the commits that fix the collected performance defect reports by searching for the IDs of performance defect reports in commit messages. An example of performance defect reports and its fixing commit is shown in Figures 4.1(a) and 4.1(b). In particular, 646 out of 2,753 (i.e., 30.4%) performance defect reports can be linked to commits by searching for issue IDs in the commit messages.

For each project, we include the performance defect fixing commits in the latest year of the development history of the project. We filter performance defect fixing commits by keeping only the commits that have source code changes, i.e., changes to **.java* files. After filtering, there is a median of 175 performance defect fixing commits in each project.

4.3.4 Identifying the files with performance defects

For each project, we exclude the test files since we are interested in studying the implications of performance defects. A file is considered as a test file if the filename contains the *test* keyword.

Figure 4.5 shows the approach that we use to label the files that have performance

defects. Following the prior studies [282, 283], we collect the source code files of each project six months before the latest commit time in the project. For example, in project Elasticsearch, the latest commit time is June 7, 2021 and we obtain the source code files of Elasticsearch on December 7, 2020 (i.e., 6 months before June 7, 2021). To identify the files that have performance defects, we make use of the commits collected in the previous step. Then, we mark the source code files that are modified in the performance defect fixing commits as performance defect files. However, there are performance defect files introduced after the time we collect the source code files. Therefore, we use SZZ [222] algorithm to search for the bug introducing commits. We exclude the buggy files that are introduced after the time that we collect the source code files. Figure 4.6 shows the number and the ratio of the files that have performance defects in our dataset. Specifically, in our projects, a median ratio of 0.84% files have performance defects, which means there is one performance defect in every 119 files. performance defects are not frequent, however, they caused many severe failures in production and resulted in software worth hundreds of millions being abandoned [186, 212].

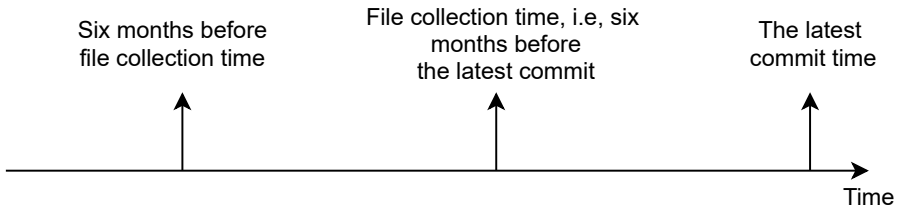
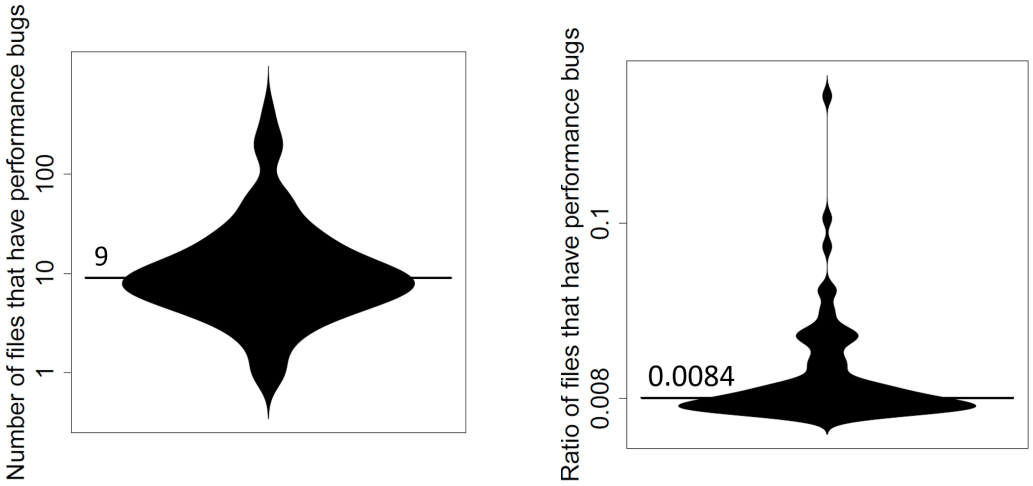


Figure 4.5: The approach that is used to label the files that have performance defects



(a) The number of the files that have performance defects in each project

(b) The ratio of the files that have performance defects in each project

Figure 4.6: The number and the ratio of the files that have performance defects in the 80 GitHub projects

4.3.5 Capturing the code characteristics of performance defects

In this section, we propose a number of metrics to capture the code characteristics of performance defects. We select four popular types of performance defects that are explored in the existing studies [53, 191, 195, 280], including performance defects with non-intrusive fixes, data corruption hang defects, redundant traversal defects, and synchronization defects. The proposed performance code metrics are shown in Table 4.3. Below, we explain the rationale of each performance code metric.

Performance defects with non-intrusive fixes. Nistor et al. [191] analyze performance defects that have *non-intrusive fixes* and find that each performance defect in the family is associated with a loop and an if-condition. For example, in the situation that an if-condition inside a running loop is met and there is no break command to exit from the loop, the loop continues its execution until an end condition is met. Such cases result to unnecessary execution and performance deterioration.

To represent the code characteristics of such performance defects, we propose the *Num_if_in_loop* and *Num_loop_in_if* metrics (as listed in Table 4.3) to count the number of if-conditions in loops and the number of loops in if-conditions. The rationale is that when loops and if-conditions are heavily nested, developers are likely to miss exit-conditions to stop the loops.

```
1 boolean elExp = False;
2 while (nodes.hasNext()) {
3       ELNode node = nodes.next();
4       if (node instanceof ELNode.Root) {
5             if (((ELNode.Root) node).getType() == '$'){
6                   elExp = true;
7                   }
8             }
9 }
```

(a) The performance defect with non-intrusive fix

```
1 boolean elExp = False;
2 while (nodes.hasNext()) {
3       + if (elExp) break; // FIX
4       ELNode node = nodes.next();
5       if (node instanceof ELNode.Root) {
6             if (((ELNode.Root) node).getType() == '$'){
7                   elExp = true;
8                   }
9             }
10 }
```

(b) The fix for the performance defect example

Figure 4.7: An example of performance defects that have non-intrusive fix and its resolution in project Tomcat [244]

Figure 4.7 shows a performance defect that has non-intrusive fix. As shown in Figure 4.7(a), when the `if-condition` at line 6 is met, the remaining computations of the loop are unnecessary because there is no point to set the `elExp` variable to true again. Such cases result in performance deterioration. Thus, developers fix the performance by adding a break statement at line 3 as shown in Figure 4.7(b).

The proposed performance code metric *Num_if_in_loop* can help us model such performance defects.

Data corruption hang bugs. Dai et al. [53] report that the *data corruption hang bugs* can cause infinite loops in software and make software unavailable to its users, which is among the most common performance issues [52, 56, 58, 139]. Moreover, data corruption hang bugs happen when a loop's exit-condition is affected by an I/O operation, e.g., reading from a file or database. For example, if a file read operation inside a loop is executed and the file is corrupted, then this can lead to an infinite loop. Therefore, we propose the *Num_file_operations*, *Num_file_operations_in_loop*, *Num_database_operations*, and *Num_database_operations_in_loop* metrics (as listed in Table 4.3) to count the number of I/O operations inside a method and loop. The rationale is that the execution of a loop is likely to be affected by the I/O operations in the loop or method.

```
78 private void readLocalFile(Path path, ...) throws
   IOException {
        ...
83      InputStream in = new FileInputStream(…);
84      byte[] data = new byte[BUFFER_SIZE];
85      long size = 0;
86      while (size >= 0){
87           size = in.read(data);
        }}
```

Figure 4.8: An example of data corruption hang performance defects in file BenchmarkThroughput.java [16] of project Hadoop Distributed File System

Figure 4.8 illustrates a data corruption hang bug in file BenchmarkThroughput.java [16] of project Hadoop Distributed File System. The data corruption hang bug is reported in issue #13514 [17]. As shown in Figure 4.8, when the `BUFFER_SIZE` variable at line 84 is 0, the `InputStream` in at line 87 reads a zero-size byte array and

returns 0. The exit-condition of the while loop at line 86 become infeasible because `size < 0` is never satisfied. The `in.read(data)` at line 87 is a file operation. Thus, the proposed performance code metrics *Num_file_operations* and *Num_file_operations_in_loop* can help us model this data corruption hang bug.

Redundant traversal bugs. Olivo et al. [195] have studied the *redundant traversal bugs* that appear when methods repeatedly iterate over a data structure, without modifying it after a successive traversal. Since a data structure is not modified between traversals, the results from one traversal can be reused and the repeatedly traversals are a waste of computational resources, which results in performance degradation. Ghanavati et al. [86] find that inefficient usage of data structures (e.g., not removing stale objects from data collections) is one of the common root causes of memory leak bugs. Inspired by the above studies [86, 195], we propose the *Num_collection* and *Num_collection_in_loop* metrics (as listed in Table 4.3) to count the number of operations on data structures. The rationale is that the extensive usage of data structures makes it challenging for developers to correctly manage them, which can lead to inefficient usages of data structures [270].

Figure 4.9 depicts a redundant traversal bug in file CandlestickRenderer.java [131] in JFreeChart. The redundant traversal bug is reported by Olivo et al. [195]. As shown in Figure 4.9, the `drawItem()` method iterates over all points in the dataset (i.e., `XYDataset dataset` at line 583) in order to draw a single data point. The `drawItem()` method traverses all data points to compute a variable called `xxWidth` at line 656. The `xxWidth` variable records the minimum gap between adjacent x-coordinates of all points in the dataset. If a dataset is not modified between successive calls to the `drawItem()` method, the re-computation of `xxWidth` in each call to

```
583 public void drawItem(XYDataset dataset, int series, ...) {
        ...
606     OHLCDataset highLowData = (OHLCDataset) dataset;
        ...
648     itemCount = highLowData.getItemCount(series);
        ...
651     for (int i=0; i< itemCount; i++) {
652          double pos = domainAxis.valueToJava2D(
653                    highLowData.getXValue(series, i), dataArea,
654                    domainEdge);
655          if (lastPos != -1) {
656               xxWidth = Math.min(xxWidth,
657                         Math.abs(pos - lastPos));
658          }
659          lastPos = pos;
        }}}
```

Figure 4.9: An example of redundant traversal performance defects in file CandlestickRenderer.java [131] of project JFreeChart

the `drawItem()` method is unnecessary. The `getXValue` at line 653 is an operation on `highLowData` which is a set object as defined by `OHLCDataset highLowData` at line 606. Thus, the proposed performance code metrics *Num_collection*, *Num_collection_in_loop*, and *Num_if_in_loop* can help us model this redundant traversal bug.

Synchronization performance defects. Thread synchronization is defined as a mechanism which ensures that two or more concurrent processes or threads do not simultaneously execute a particular program segment known as critical section. However, synchronization performance defects can cause synchronization issues and degrade the performance of a multi-threaded system. To this end, Zhang et al. [280] investigate synchronization performance defects and find that the nested loops in critical sections are likely to cause synchronization issues. This is because nested loops are time-consuming and once a thread enters the critical section and starts to execute the nested loops, then all other threads have to wait for the nested loops to

finish. Therefore, we propose the *Num_nested_loop* and *Num_nested_loop_in_crit* metrics (as listed in Table 4.3) to count the number of nested loops found in methods' critical sections. In addition, we propose the *Num_synchronization* and *Num_thread* metrics (as listed in Table 4.3) to represent the number of synchronization operations (e.g., acquiring a mutex) and thread operations (e.g., creating a thread) in a method. The rationale is that a method is more likely to have synchronization issues if it has more synchronization and thread operations.

```
993   public void handle(JobEvent event) {
          ...
998     try {
999                 writeLock.lock();
                    ...
1002                doTransition(event.getType(), event);
1003      } catch
              ...
1018    finally {
1019      writeLock.unlock();
      }}
```

Figure 4.10: An example of synchronization performance defects in file JobImpl.java [18] of project Hadoop MapReduce

Figure 4.10 shows a synchronization bug in file JobImpl.java [18] of project Hadoop MapReduce. The synchronization bug is reported in issue #4813. As shown in Figure 4.10, the `handle()` method acquires the `writeLock` lock at line 999, executes the method `doTransition()` at line 1002, and releases the `writeLock` lock at line 1019. However, if the method `doTransition()` takes too long to finish, some threads in the system have to wait for the release of the `writeLock` lock and the system might become unresponsive to users. The code conducts synchronization operations (e.g., acquiring and releasing a lock). Thus, the proposed performance code metric *Num_synchronization* can be used to model such synchronization bug.

The proposed performance code metrics can be calculated via static source code analysis and require no system specific knowledge. We focus on loops when we propose the performance code metrics because most computation time is spent inside loops and most performance defects involve loops [103, 193, 267, 278].

Prior studies [44, 55, 62] propose metrics to capture the performance regression or improvement introduced by source code changes. Compared with the metrics proposed in the prior studies [44, 55, 62], the proposed performance code metrics in this chapter are oriented to capture the code characteristics of performance defects. For example, the prior studies [44, 55, 62] count only the number of loops in a method, while we count (1) the number of loops in if-conditions, (2) the number of file operations in loops, and (3) the number of database operations in loops to capture the code characteristics of data corruption hang bugs [53].

4.3.6 Calculating the performance code metrics

To calculate the performance code metrics of a method, we parse a method to identify various code structures including, the loops and if-conditions, and the file, database, data structure, and synchronization objects within the method. From the identified objects, we can count the number of operations performed on the corresponding objects.

Parsing the source code. We parse the source code of a method to build Abstract Syntax Tree (AST). The AST represents the abstract syntactic structure of the source code. Figures 4.11(a) and 4.11(b) depict the source code of an example method, named *setForcedRefresh*, in file BulkShardResponse.java [67] in Elasticsearch and the AST of the method. As shown in Figure 4.11(b), the AST identifies that

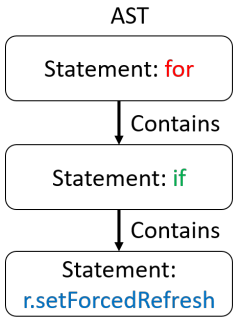
Table 4.3: The proposed performance code metrics. The metrics are aggregated to a file level using average scheme.

Performance code metric	**Description**
Num_if_in_loop	Number of if-conditions that are inside loops
Num_loop_in_if	Number of loops that are inside if-conditions
Num_file_operations	Number of file operations
Num_file_operations _in_loop	Number of file operations inside loops
Num_database_operations	Number of databases operations
Num_database_operations_in _loop	Number of databases operations inside loops
Num_collection	Number of operations on collection data structures
Num_collection_in _loop	Number of operations on collection data structure inside loops
Num_synchronization	Number of synchronization operations
Num_nested_loop	Number of nested loops
Num_nested_loop_in _crit	Number of nested loops in critical sections in synchronization
Num_thread	Number of thread operations

there is a for-loop and an if-condition inside the scope of the for-loop. We use the ANTLR (ANother Tool for Language Recognition) [201] to build the ASTs of all files of the selected Java projects. ANTLR builds AST from Java source code based on the grammar of the Java language [11]. ANTLR is a powerful parser generator for reading and processing source code and has already been used in prior studies [128, 202, 254].

```
public void setForcedRefresh(boolean forcedRefresh) {
    ...
    for (BulkItemResponse response : responses) {
        DocWriteResponse r = response.getResponse();
        if (r != null) {
            r.setForcedRefresh(forcedRefresh);
        }
    }
}
```

(a) The source code of the setForcedRefresh method



(b) The AST

Figure 4.11: The source code and AST of the *setForcedRefresh* method in Elasticsearch.

Identifying objects in source code. We use a commercial tool, called Understand [1], to count the objects (i.e., source code artifacts) in a method. Table 4.4 illustrates the objects in the *setForcedRefresh* method (see Figure 4.11(a)). As shown in Table 4.4, the Understand tool is able to identify the objects and the type of the objects in a method. To calculate the performance code metrics, we need to distinguish five object categories of code artifacts (i.e., file, database, data structure, synchronization, and thread). We use the IO, data structures, database access, thread management, and synchronization classes provided by the JDK library to classify the types of objects. In addition, developers in a project can create their own object types to fit the needs of a project. For example, developers in Elasticsearch define the *DocWriteResponse* object type, as shown in Figure 4.11(a), to conduct a write operation on a single document. The *DocWriteResponse* object type does not inherit from any Java file operation class [70]. It is challenging to manually go through each project and classify all the defined objects into the five categories. Therefore, we define a number of keywords to classify objects into the five above-mentioned categories (see Table 4.5). For example, object r has the *DocWriteResponse* as its object

type (see in Table 4.4) which contains the *write* keyword. Therefore, we classify the *r* object under the file object category since the *write* keyword belongs there (see Table 4.5).

In total, 7,495,049 objects are returned by Understand tool after analyzing the source code of our projects. To understand the accuracy of our heuristic approach in classifying the types of objects, we conduct a manual analysis following the steps mentioned in Section 4.3.2. We randomly select a sample of 340 objects using 95% confidence level and 5% confidence interval. Then, we manually label the types of the sampled objects. We compare the labels of the co-authors and obtain a 0.84 Cohen's Kappa score, which indicates almost perfect agreement between the two co-authors [179]. Through the manual analysis, we find that our approach is able to correctly classify the types for 322 objects out of the 340 objects (i.e., 322/340 = 94.8%). Among the 322 objects that are correctly classified, 56 objects are classified by using the class names in the JDK library.

Table 4.4: The objects of *setForcedRefresh* method

Object name	**Object type**
forcedRefresh	boolean
responses	Array of BulkItemResponse
response	BulkItemResponse
r	DocWriteResponse

Identifying objects' operations. We traverse the obtained ASTs to count the number of operations on the identified objects. For example, as shown in Figure 4.11(b), we traverse the AST of the *setForcedRefresh* method and identify the *r.setForcedRefresh()* statement as an operation on the file object *r*.

In total, we calculate the performance code metrics of 1,342,896 methods from the

Table 4.5: The keywords that are used to identify each category of objects. The performance code metrics column represents the metrics that are calculated using each category of objects.

Category of object	**Keyword for the category**	**Performance code metrics**
File objects	file, write, read, stream	Num_file_operations, Num_file_opertaions _in_loop
Database objects	database, db, statement	Num_database _operations, Num_database _operations _in_loop
Data structure objects	collection, set, iterator, list, dequeue, map, hash, array, queue, vector	Num_collection, Num_collection _in_loop
Synchroni-zation objects	semaphore, lock	Num_synchroni-zation, Num_nested _loop_in_crit
Thread objects	thread	Num_thread

collected projects. The performance code metrics are calculated at the method level to measure the code characteristics of performance defects. Our performance defect prediction is performed at a file level in this chapter. Similar to the prior study [281], we aggregate the performance code metrics to a file level using the average aggregation scheme, e.g., the value of the *Num_nested_loop* metric of a file is the average value of the *Num_nested_loop* metric of each method in the file.

4.3.7 Calculating the code and process metrics

To represent the source code and code change history, we include the code and process metrics that are widely used in the prior defect prediction studies [19, 22, 31, 64, 108, 161, 181, 281, 283, 284]. Table 4.6 summarizes the code and process metrics.

Code metrics. For each project, we calculate the code metrics by analyzing the collected source code files six months before the latest commit time in the project. We use the Understand tool [1] to compute the code metrics (see Table 4.6) [5, 281, 283]. Following the existing study [281], we aggregate the code metrics to a file level using the average aggregation scheme, e.g., the value of the LOC metric of a file is the average value of the LOC metric of each method in the file.

Process metrics. The process metrics are calculated using the code change history in the six months period before the time that we collect source code files as shown in Figure 4.5. We implement scripts to automatically query the history revisions (i.e., commits) of a file and identify the performance defect fixing commits in the commit history. A commit is considered as a bug fixing commit if its message contains any of the following keywords: *bug*, *fix*, *error*, *issue*, *crash*, *problem*, *fail*, *defect*, and *patch* [5, 281, 283]. To count the number of bugs appeared in a file before the file collection time, we identify the bug IDs in the commit messages (see Figure 4.1(b)). The number of the added and deleted lines in each commit of a file before the file collection time is collected and aggregated to a file level using the average value.

Table 4.6: The code and process metrics used in this chapter. The last column refers to the scheme to aggregate method level metrics to a file level ("none means that no aggregation is performed for metrics that are calculated at a file level").

Metric type	**Metric name**	**Description**	**Aggregation scheme**
Code metrics	LOC	Lines of code in a method	average
	CL	Comment lines in a method	average
	NSTMT	Number of statements in a method	average
	RCC	Ratio comments to codes of a method	average
	MNL	Max nesting level of a method	average
	CC	McCable cyclomatic complexity of a method	average
	FANIN	Number of input data of a method	average
	FANOUT	Number of output data of a method	average
Process metrics	Num_rev	Number of revisions	None
	Num_perf_rev	Number of revisions a file was involved in fixing performance defects	None
	Num_non_perf_rev	Number of revisions a file was involved in fixing non-performance defects	None
	Num_perf_bug	Number of performance defects happened in a file	None
	Num_non_perf_bug	Number of non-performance defects happened in a file	None
	Added_loc	Lines of code added in a file in the history commits	average
	Deleted_loc	Lines of code deleted in a file in the history commits	average

4.4 Experiment Results

In this section, we present motivations, approaches, and results of the studied research questions.

4.4.1 RQ1. What is the performance of our approach in predicting performance defects at file level?

Motivation: Prior studies [53, 86, 191, 195, 270, 280] have proposed different approaches to find performance defects via identifying anti-patterns in source code. However, each type of performance defect has its own unique anti-pattern and requires a different approach to analyze the source code. Adopting a different approach to detect each type of performance defect is time-consuming. Moreover, prior approaches [53, 86, 191, 195, 280] cannot predict performance defects that do not follow the identified anti-patterns. To help developers in identifying performance defects at an early stage of the development phase, we provide a unified approach that can predict various types of performance defects. Specifically, we combine the proposed performance code metrics with the code and process metrics used in defect prediction studies [19, 22, 31, 64, 108, 161, 181, 281, 283, 284] to build models for predicting performance defects at file level.

To answer this RQ, we use the performance code metrics proposed in Section 4.3 (see Table 4.3). As shown in Figure 4.4, after calculating the metrics and labeling the files using the automatic approaches mentioned in Section 4.3, we build machine learning models and evaluate their performance of predicting performance defects.

Selecting machine learning algorithms. Various machine learning algorithms have been applied to predict defects in software systems [238]. In our experiment,

we select machine learning algorithms that are widely used to predict defects in the existing studies [5, 238, 283, 284]. To this end, seven machine learning algorithms, including Random Forest (RF), eXtreme Gradient Boosting (XGBoost), Logistic Regression (LR), Support Vector Machines (SVM), Complement Naive Bayes (CNB), MultiLayer Perceptron neural network (MLP), and Decision Tree (DT), are selected.

Creating training and testing datasets. In this chapter, the performance defect prediction models are within-project models. A model is a within-project performance defect prediction model if both training and testing datasets are from the same project [282]. We apply the out-of-sample bootstrap technique [5, 65, 237, 238] to create training and testing datasets for each project. Out-of-sample bootstrap leverages aspects of statistical inference [66] and is widely used to obtain robust evaluation results of prediction models. Prior studies [237, 238] observe that the out-of-sample bootstrap can produce the less biased and more stable performance estimates than the k-fold cross-validation. Moreover, the out-of-sample bootstrap is recommended for highly-skewed datasets [111, 238], which is the case in performance defect prediction where the number of the files that have performance defects is small comparing with clean files. As shown in Figure 4.6(b), the ratio of the files that have performance defects is lower than 1%. The out-of-sample bootstrap is composed from two steps:

- For a project with N files, a bootstrap sample of size N files is randomly drawn with replacement from the original files of the project.
- A model is trained and validated using the bootstrap sample and tested using the files that do not appear in the bootstrap sample. On average, 36.8% of the files do not appear in the bootstrap sample, since the sample is drawn with

replacement [65].

We repeat the out-of-sample bootstrap process for 100 times to create 100 training and testing datasets for each project.

Conducting class re-balancing on the training datasets. As suggested by the prior study [238], class re-balancing is able to improve the performance of defect prediction models. As shown in Figure 4.6(b), our dataset is very imbalanced as there is only a median of 0.84% files that have performance defects in our dataset. The prior study [238] observes that the SMOTE class re-balancing technique [42] improves the performance of defect prediction models the most. Thus, we use the SMOTE class re-balancing to balance the number of files that have performance defects in our training datasets.

Performance metrics. We use three metrics to evaluate the performance of our models, including Area Under the receiver operator characteristic Curve (AUC), Matthews Correlation Coefficient (MCC), and Area under the Precision-Recall curve (PR-AUC). The AUC is calculated by measuring the area under the curve that plots the true positive rates against the false positive rates. The values of the AUC range from 0 (inverse prediction) to 1 (perfect prediction), where a 0.5 value means that a model randomly guesses if a file has performance defects. Similarly, the PR-AUC measures the area under the curve that plots the precision against recall. The PR-AUC ranges from 0 (worst performance) to 1 (best performance). We use AUC and PR-AUC because AUC and PR-AUC do not require the threshold calculation compared to threshold-dependent performance metrics (e.g., accuracy and F-measure). In addition, we use MCC because it is more suitable for imbalanced datasets like ours [48]. Compared with AUC and PR-AUC, MCC produces high scores only if the

predictions obtain good results in all of the four confusion matrix categories (i.e., true positive, false positive, true negatives, and false positives) [48]. MCC measures the differences between the actual labels and the predicted labels. Moreover, it ranges from -1 (inverse prediction) to 1 (perfect prediction), while 0 means random prediction.

Training performance defect prediction models. To find the best hyper-parameters for our models and the SMOTE class re-balancing technique, we use the grid search optimization approach. The grid search parameter optimization approach consists of three steps.

- **Set candidate values for parameters**. We manually input a set of candidate values for every parameter in each machine learning algorithm and the SMOTE technique.

- **Iterate candidate parameter values**. We iterate all possible combinations of candidate parameter values. For example, if a machine learning algorithm has two parameters and each parameter has 5 candidate values, then we obtain 25 (i.e., 5*5) possible combinations of candidate parameter values. For each possible combination, we train models and test their performances. generated from the out-of-sample bootstrap process, we train our model using 80% of the training dataset and test them on the rest 20%.

- **Select the optimal parameter values**. After iterating all the combinations of parameters, we select the ones that achieves the highest performance.

Evaluating the performance defect prediction models. After selecting the optimal parameter values, we evaluate the performance of the machine learning

algorithms on the testing datasets. In a project, for each training and testing datasets that are generated from the out-of-sample bootstrap process, we train a machine learning model on the training dataset using the optimal parameter values and test the AUC, PR-AUC, and MCC of the model for predicting performance defects in the testing dataset. Since we have seven machine learning algorithms, 80 projects, and 100 datasets from each project, we build and test 56,000 (i.e., 7*80*100) models, in total.

Comparing the performance of the models. After evaluating the performance of machine learning models on our projects, we draw beanplots of the AUCs, PR-AUCs, and MCCs to visualize the performance of the models. Next, we conduct Friedman tests followed by Nemenyis post-hoc tests to compare the performance of different models. Both Friedman test and Nemenyi's post-hoc test are non-parametric tests that do not require the analyzed data to meet any assumptions [205].

We conduct a Friedman test followed by a Nemenyis post-hoc test on each performance metric (i.e., AUC, PR-AUC, and MCC) measured from evaluating machine learning algorithms. For example, we conduct a Friedman test on AUCs to test whether the AUCs of the studied machine learning models are significantly different. If the Friedman test suggests that there are significant differences among the AUCs of the seven machine learning algorithms, we then use the Nemenyis post-hoc test to compare the AUCs of each pair of machine learning algorithms. If the AUCs of two algorithms are significantly different, we classify the two algorithms into different performance groups. Otherwise, we classify the two algorithms into the same performance group. We use 0.01 as the significant level when applying Friedman tests and Nemenyi's post-hoc tests.

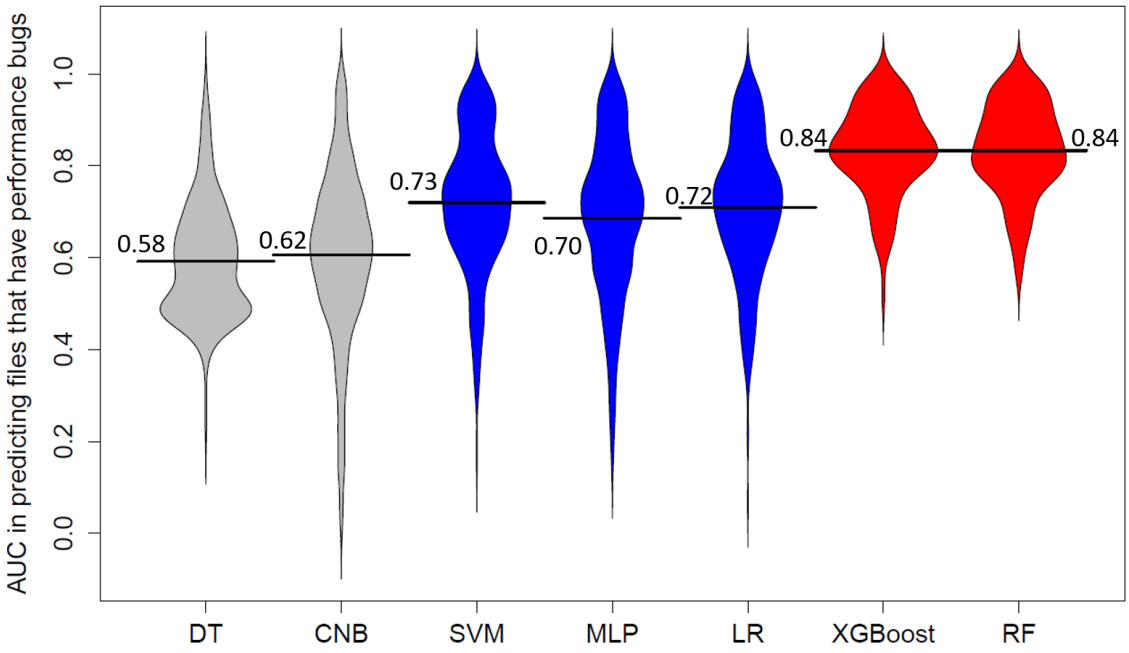


Figure 4.12: The AUC of machine learning algorithms for predicting performance defects at file level

Table 4.7: The p-values of the Friedman tests of comparing the performance of machine learning algorithms

Performance metrics		
MCC	**PR-AUC**	**AUC**
5.7e-78	2.0e-137	1.5e-47

Table 4.8: The classified groups based on the results from Nemenyi's post-hoc tests

Performance metrics	**Groups assigned based on the results from Nemenyi's post-hoc tests**		
	First group	**Second group**	**Third group**
MCC	RF, XGBoost	LR, MLP, SVM, CNB	DT
PR-AUC	RF, XGBoost	LR, MLP, SVM, CNB, DT	None
AUC	RF, XGBoost	LR, MLP, SVM, CNB	CNB, DT

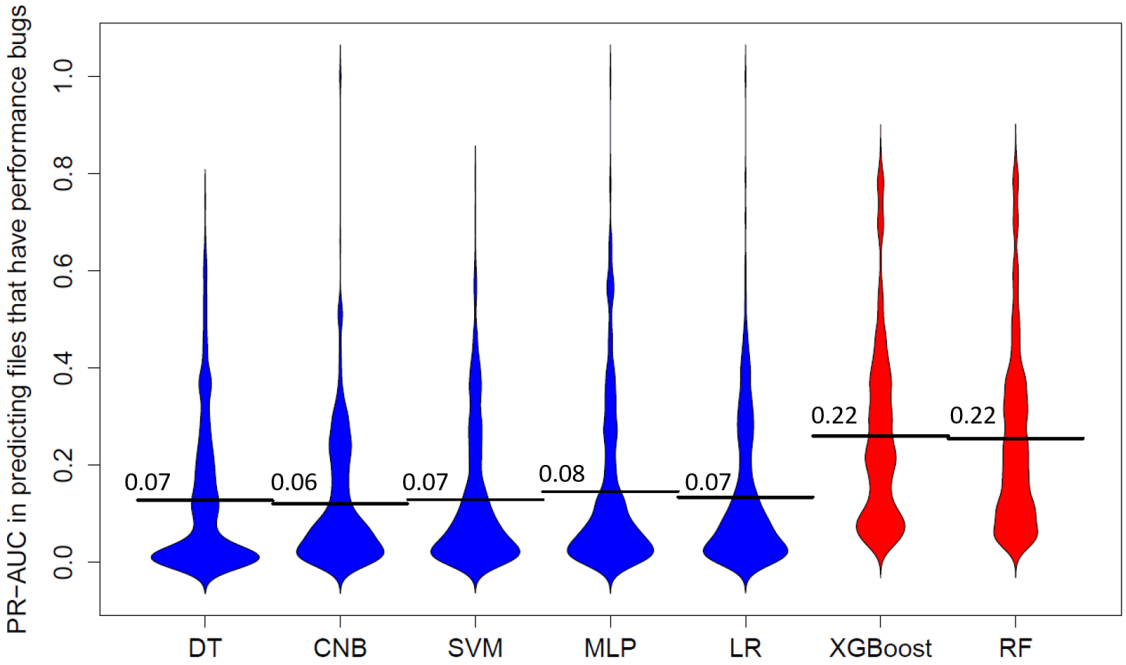


Figure 4.13: The PR-AUC of machine learning algorithms for predicting performance defects at file level

Results: The Random Forest and eXtreme Gradient Boosting algorithms achieve the best performance in predicting files with performance defects. Tables 4.7 and 4.8 show the results of Friedman tests and Nemenyi's post-hoc tests of comparing the performance of seven machine learning algorithms. As shown in Table 4.7, the p-values of the Friedman tests are all smaller than 0.01, which means that the performance of the studied machine learning algorithms are significantly different under all performance metrics. Table 4.8 shows the groups of machine learning algorithms based on the pairwise comparisons from Nemenyi's post-hoc tests. As shown in Table 4.8, the Random Forest and eXtreme Gradient Boosting algorithms are consistently ranked in the best performing group (i.e., first group).

Figures 4.12, 4.13, and 4.14 show the AUCs, PR-AUCs, and MCCs of the studied

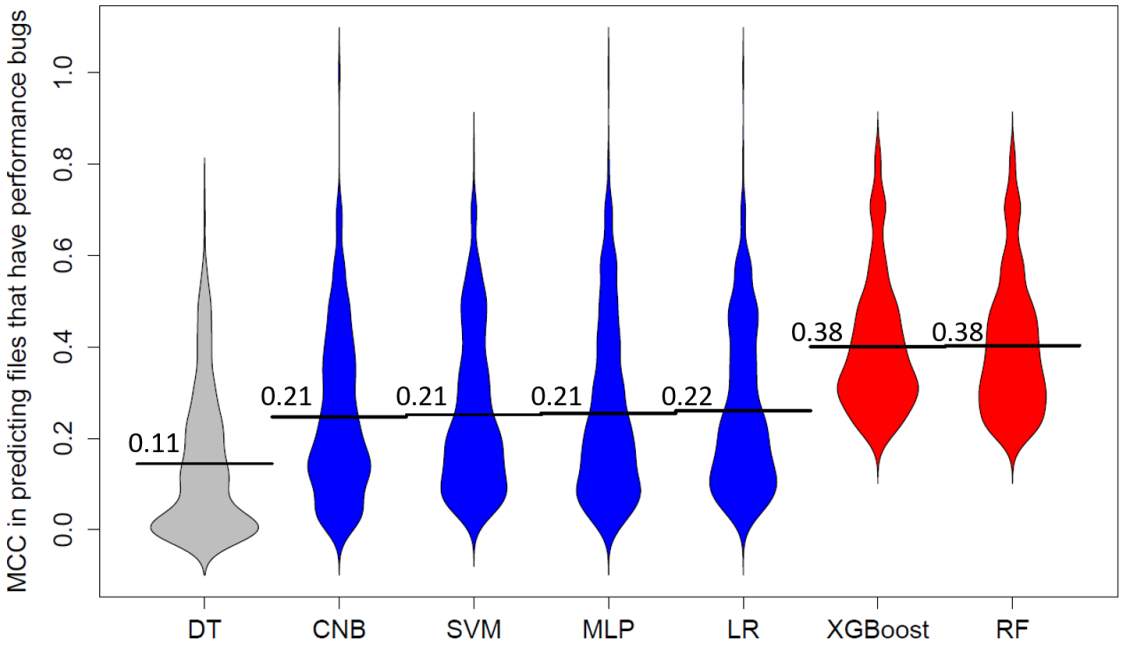


Figure 4.14: The MCC of machine learning algorithms for predicting performance defects at file level

algorithms for predicting files that have performance defects. The colors in Figures 4.12, 4.13, and 4.14 represent the groups of machine learning algorithms ranked by the Nemenyi's post-hoc tests. As shown in Figures 4.12, 4.13, and 4.14, the Random Forest and eXtreme Gradient Boosting algorithms achieve the best performance with a median of 0.38 MCC, 0.22 PR-AUC, and 0.84 AUC. The 0.84 median AUC value (ranges from 0 to 1) suggests that the algorithms achieve a high quality of performance for distinguishing the files with performance defects from the files without performance defects. As suggested by the existing studies [35, 48, 224], the median 0.22 PR-AUC (ranges from 0 to 1) and 0.38 MCC (ranges from -1 to 1) suggest relatively mediocre performance. Overall, the Random Forest and eXtreme Gradient Boosting algorithms achieve acceptable performance for predicting performance defects at the file level. One explanation to the mediocre PR-AUC and MCC values

is that the ratio of source code files that have performance defects is very low, i.e., 0.84%, in our studied projects. It is a challenging task to train machine learning algorithms to predict performance defects with such a low ratio.

The Complement Naive Bayes and Decision Tree algorithms have the worst performance for predicting performance defects. A possible reason is that the Complement Naive Bayes and Decision Tree algorithms are not suitable for predicting highly-skewed performance defects, since there is a very small number of performance defect files (i.e., a median of 0.84% as shown in Figure 4.6(b)).

Summary

Among the examined algorithms, the Random Forest and eXtreme Gradient Boosting algorithms achieve the best performance in predicting performance defects at file level.

4.4.2 RQ2. Which group of metrics affect the performance of our models the most?

Motivation: To build performance defect prediction models, we use three groups of metrics including code metrics, process metrics, and the proposed performance code metrics. To understand whether the proposed performance code metrics are indeed useful in predicting performance defects, we analyze the effects that each group of metrics has on the prediction models.

Approach: To test the effect of a group of metrics on the machine learning models, we apply the effect calculation process as suggested by the prior study [238]. The calculation of the effect of each group of metrics on a machine learning algorithm is made of two steps:

- For each dataset (i.e., the files and their metrics), we first randomly permute all the values of a group of metrics and obtain a new dataset.

- We apply out-of-sample bootstrap process on the new dataset and conduct the parameter optimization to select the optimal parameter values for the machine learning algorithm. Next, we evaluate the performance of the machine learning algorithm for predicting performance defects on the new dataset. We compute the differences between the performance of the models that are built on the original dataset and the dataset with the randomly-permuted metrics. The performance differences are measured in AUC, PR-AUC, and MCC and are used as the effect of the group of metrics.

We follow the same steps to test the effects of all groups of metrics on all studied machine learning algorithms except the Decision Trees and Complement Naive Bayes algorithms. We did not include the Decision Trees and Complement Naive Bayes algorithms in the metrics effects study because these two algorithms have poor performance for predicting performance defects as discussed in Section 4.4.1. After measuring the effects of all groups of metrics on the rest of the five algorithms, we apply the Friedman tests followed by Nemenyi's post-hoc tests to compare the effects of all groups of metrics.

Results: The proposed performance code metrics impact the performance of our models the most. Tables 4.10 and 4.11 show the results of Friedman tests and Nemenyi's post-hoc tests of comparing the effects of the groups of metrics. As shown in Table 4.10, the p-values of the Friedman tests are all smaller than 0.01, which means that the effects of groups of metrics are significantly different under all performance metrics. Table 4.11 shows the groups of metrics ranked based on the

Table 4.9: The effects of groups of metrics on performance defect prediction models. We highlight the highest effects of groups of metrics in each algorithm.

Groups of metrics		**Median of performance drop percentage**				
		RF	**XGB**	**LR**	**MLP**	**SVM**
Performance code metrics	MCC	**19.0%**	**18.5%**	**20.3%**	**20.0%**	13.2%
	PR-AUC	**27.2%**	**24.7%**	13.3%	**23.2%**	14.3%
	AUC	**7.0%**	**6.9%**	**9.2%**	**11.2%**	**5.5%**
Code metrics	MCC	5.9%	4.3%	3.1%	5.7%	4.1%
	PR-AUC	10.0%	10.4%	7.9%	0.1%	6.6%
	AUC	6.2%	6.7%	1.4%	0.1%	0.2%
Process metrics	MCC	7.8%	4.4%	14.2%	2.7%	**18.0%**
	PR-AUC	11.9%	8.1%	**17.7%**	10.6%	**22.7%**
	AUC	1.3%	1.2%	2.7%	1.9%	3.0%

Table 4.10: The p-values of the Friedman tests of comparing the effects of groups of metrics

Performance metrics		
MCC	**PR-AUC**	**AUC**
4.7e-11	6.3e-12	5.7e-08

Table 4.11: The classified groups based on the results from Nemenyi's post-hoc tests

Performance metrics	**Groups assigned based on the results from Nemenyi's post-hoc tests**		
	First group	**Second group**	**Third group**
MCC	Performance code metrics	Process metrics	Code metrics
PR-AUC	Performance code metrics	Process metrics	Code metrics
AUC	Performance code metrics	Code metrics	Process metrics

pairwise comparisons from Nemenyi's post-hoc tests. As shown in Table 4.11, the proposed performance code metrics are consistently ranked in the best performing group (i.e., first group).

Table 4.9 shows the effect that each group of metrics has on the five studied machine learning algorithms, measured in AUC, PR-AUC, and MCC. As shown in Table 4.9, the proposed performance code metrics have the highest effects for our models. For example, building a Random Forest model without the proposed performance code metrics can reduce the model's median AUC, PR-AUC, and MCC by 7.0%, 27.2%, and 19.0%, respectively. On average, the AUC, PR-AUC, and MCC of the five studied machine learning models drop a median of 7.7%, 25.4%, and 20.2% without using the proposed performance code metrics.

In addition to studying the effects of groups of metrics, we measure the importance of each metric on the studied machine learning algorithms as shown. Table 4.12 shows the effects of metrics on performance defect prediction models. We find that the top two most important metrics are the proposed performance code metrics (i.e., Num_collection and Num_file_operations). As shown in Table 4.12, 7 out of the top 10 most important metrics are proposed performance code metrics (i.e., Num_collection, Num_file_operations, Num_thread, Num_synchronization, Num_database_operations, Num_if_in_loop, and Num_collection_in_loop), while only 2 of the code metrics (i.e., RCC and FANOUT) and 1 of the process metrics (i.e., Num_non_perf_rev) are ranked in the top 10.

Table 4.12: The effects of metrics on performance defect prediction models

Metric	Average value of effects
Num_collection	0.037
Num_file_operations	0.036
RCC	0.031
FANOUT	0.030
Num_synchronization	0.027
Num_database_operations	0.026
Num_non_perf_rev	0.022
Num_thread	0.019
Num_if_in_loop	0.016
Num_collection_in_loop	0.015
Num_rev	0.015
CL	0.015
Num_loop_in_if	0.014
Num_file_operations_in_loop	0.013
Num_nested_loop	0.013
CC	0.012
Added_loc	0.011
FANIN	0.009
Num_database_operations_in_loop	0.008
Num_nested_loop_in_crit	0.008
Num_perf_rev	0.007
Deleted_loc	0.003
MNL	0.002
Num_perf_bug	0.001
Num_non_perf_bug	0.001
NSTMT	0.001
LOC	0.000

Summary

The proposed performance code metrics impact the performance of our prediction models the most. Specifically, the AUC, PR-AUC, and MCC of the five studied machine learning models drop a median of 7.7%, 25.4%, and 20.2% without using the proposed performance code metrics.

4.4.3 RQ3. What are the different types of performance defects that our approach can predict?

Motivation: In this chapter, we aim to provide a unified approach that can predict various types of performance defects by combining the performance code metrics with code and process metrics. To understand the types of performance defects that our approach can predict, we analyze the performance defects predicted by our approach.

Approach: To answer this RQ, we first collect performance defects that are predicted by our approach. Then, we manually go through a statistical sample of the predicted performance defects and analyze the types of the predicted performance defects.

Collecting performance defects predicted by our approach. First, we build Random Forest models using its optimal parameters values and training datasets selected in the experiments in Section 4.4.1. Then, we use the Random Forest models to predict files that have performance defects in the testing datasets (see Section 4.4.1) from the 80 experiment projects. Random Forest model predicts the probabilities of files to have performance defects and we use 0.5 as the threshold to convert the probabilities to labels (i.e., clean files and files containing performance defects). 31,556 files are predicted to have performance defects in the testing datasets from the 80 experiment projects.

Analyzing the predicted performance defects. Following the steps of manual analysis mentioned in Section 4.3.2, we randomly select a sample of 340 predicted to have performance defects. Next, we find the performance defect fixing commits for fixing the performance defects in the sampled files. Then, the two co-authors manually go through the files and performance defect fixing commits to study the

types of the performance defects. We compare the results from the two co-authors and obtain a 0.85 Cohen's Kappa score (i.e., a almost perfect agreement between two co-authors [179]).

Table 4.13: The result of the manual analysis about the types of performance defects predicted by our approach

Types of performance defects	**Number of files**	**Percentage of files**
Performance regression bug	236	69.4%
Memory leak bug	26	7.6%
Infinite loop bug	23	6.8%
Deadlock bug	21	6.2%
Stuck/hang bug	21	6.2%
Non-performance defect	13	3.8%

Results: Our approach can predict various types of performance defects. The result of the manual analysis is shown in Table 4.13. Our approach predicts performance regression bugs, memory leak bugs, infinite loop bugs, deadlock bugs, and hang bugs. As shown in Table 4.13, 236 (i.e., 236/340 = 69.4%) files contain inefficient code that results in performance regression bugs. For example, file *PublicationTransportHandler.java* [71] in project Elasticsearch causes significant system response time degradation and is predicted to have performance defects by our approach. The performance regression bug is fixed in commit c5315744 [69]. There are 13 (i.e., 236/340 = 3.8%) false positive predictions, i.e., non-performance defects, made by our approach as shown in Table 4.13. From analyzing the files that have memory leak bugs, deadlock bugs, and hang bugs, we find that our approach can predict additional performance defects that are not covered by the anti-patterns proposed in the prior studies [53, 86, 191, 195, 280]. We show examples of predicted buggy files that have memory leak bugs, deadlock bugs, and hang bugs as follows.

- The predicted buggy file *JournaledGroup.java* [4] in project Alluxio contains a memory leak bug. The memory leak bug is caused by unclosed threats when users iterate directories (detailed description can be found in commit 58c24eb8 [3]). The memory leak bug does not follow the memory leak anti-pattern (i.e., inefficient usage of data structures) proposed in the prior studies [53, 86, 191, 195, 280].

- The predicted buggy file *ConnectivityService.java* [10] in project platform_frameworks_base contains a deadlock bug. The deadlock bug is caused by poor designed lock acquiring sequences (detailed description can be found in commit 465088ed [9]). The deadlock bug does not follow the anti-pattern (i.e., nested loops in critical sections) of synchronization bugs proposed in the prior studies [53, 86, 191, 195, 280].

- The predicted buggy file *AbstractBuilding.java* [158] in project minecolonies has a hang bug that is caused by keeping searching for objects that do not exist (mentioned in commits 508e7638 [159]). The hang bug does not follow the anti-pattern (i.e., exit-conditions of loops are affected by I/O operations) proposed in the prior studies [53, 86, 191, 195, 280].

Our analysis suggests that our approach is able to capture more performance defects than the prior studies [53, 86, 191, 195, 280] because we measure code characteristics that lead to performance defects instead of measuring the restricted rules. In addition, we combine performance code metrics with code and process metrics that are extracted from the software development history to predict performance defects.

Summary

Our approach can predict various types of performance defects, including performance regression bugs, memory leak bugs, infinite loop bugs, deadlock bugs, and stuck/hang bugs. Compared to prior studies [53, 86, 191, 195, 280], our approach is able to predict additional performance defects that are not covered by the anti-patterns proposed in these prior studies.

4.5 Discussion

In this section, we discuss the challenges in fixing performance defects, the usage of our approach, and future research directions based on our approach.

4.5.1 Challenges in fixing performance defects

To investigate the reasons *why performance defects are time-consuming to fix*, we conduct a manual analysis following the steps shown in Section 4.3.2. We re-use the sample of 340 performance defect reports that are selected in Section 4.3.3. The two co-authors go through each issue report and summarize the reasons why the performance defect is time-consuming to fix. We compare the results of the two authors and obtain a Cohen's Kappa score of 0.82, which indicates a almost perfect agreement between two authors [179].

We observe that fixing performance defects requires a median of 188 hours and two developers. The challenges in fixing performance defects obtained through the manual analysis is shown in Table 4.14.

- *Addressed quickly.* There are 47 (i.e., 47/220 = 21.4%) performance defect reports that are fixed quickly after being submitted. For example, issue #127 [182]

was fixed by developers within two hours by adding one line of code after its submission. Thus, these 47 performance defect reports are not analyzed further in this chapter.

- *Difficulties in reproducing the bug.* A common reason of why it is hard to fix a performance defect is the difficulty to reproduce it. There are 32 (i.e., 32/220 = 14.5%) performance defect reports where developers encounter difficulties or are unable to reproduce them. For example, issue #5104 [109] in project *hazelcast* could not be reproduced by developers and have to be closed. Nevertheless, for mobile device applications, such as *Exoplayer* (an Android-based media player system), its developers struggled to reproduce the posted performance defect issues due to the variety of mobile devices that have different configurations, hardware, and customized Android firmwares. In addition, some performance defect issue authors do not take into account the guidelines to facilitate the reproduction process and developers have to spend time asking for additional information to reproduce the bugs.

- *Difficult to analyze the root causes of the bug.* For 27 (i.e., 27/220 = 12.3%) performance defects, we find that it is difficult to analyze their root causes. For example, developers observed memory leak when reproducing issue #2384 [209] in the project *netty*, however, they could not figure out the reasons behind this issue.

- *Long time in discussing how to fix the bug.* For 38 (i.e., 27/220 = 17.3%) performance defects, developers spend long time in discussing the possible fixes of the performance defects. For example, developers spent four days in discussing the

solutions to optimize the performance regression mentioned in issue #2468 [33] in project *ImmersiveEngineering*.

- *Developers have to submit multiple fixes.* There are 24 (i.e., 27/220 = 10.9%) performance defects where developers submit multiple pull requests to fix them. For instance, developers submit six pull requests to fix a performance regression introduced by Java trace instrument in issue #226 [197] of project *open-liberty*.

- *Low priorities in fixing the bug.* For 30 (i.e., 27/220 = 13.6%) performance defects, developers delayed fixes for the corresponding bugs by giving them low priorities and assigning them as future milestones. For example, developers first assigned issue #2309 [208] in project *scheduling* into the future development milestone and started working on a fix six months after its submission.

- *Wait for answers from expert developers.* For 7 (i.e., 27/220 = 3.2%) performance defect reports, developers had to wait for an answer or confirmation from expert developers.

- *Other reasons.* There are other challenges in fixing performance defects, such as waiting for fixes of external software and waiting for response from users. For instance, developers in project *snow-owl* had to wait for updates in project *Elasticsearch* when addressing the performance regression in issue #626 [110].

Overall, we observe that it is difficult to reproduce performance defects, analyze their root causes, and recommend fixes.

Table 4.14: The challenges in fixing performance defects

Reasons	**Number of issue reports**	**Percentage of issue reports**
Addressed quickly	47	21.4%
Long time in discussing how to fix the bug	38	17.3%
Difficulties in reproducing the bug	32	14.5%
Low priority in fixing this bug	30	13.6%
Difficult to analyze the root causes of the bug	27	12.3%
Developers have to submit multiple fixes	24	10.9%
Wait for answer from expert developers	7	3.2%
Other reasons	15	6.8%

4.5.2 Limitations of our approach

To understand the limitation of our approach, we conduct a manual analysis of the false negatives (i.e., files with performance defects that our approach fails to predict) from the prediction results of our approach. We select a sample of 340 false negatives using 95% confidence level and 5% confidence interval from the prediction results of the Random Forest models obtained from in Section 4.4.3. The Random Forest models do not make any false positives (i.e., clean files that are misclassified as files with performance defects) when using 0.5 as the threshold (please see Section 4.4.3). Thus, We do not conduct manual analysis of the false positives. Through the manual analysis, we find the performance defect fixing commits for fixing the performance defects in the sampled false negative files. Then, we go through the files and performance defect fixing commits to study the types of the performance defects and the

root causes of the performance defects. The result of the manual analysis is shown in Table 4.15.

Table 4.15: The result of the manual analysis on the types of performance defects that our approach fails to predict

Type of performance defects	**Number of files**	**Main root causes**
Performance regression bug	251	Inefficient functional logic, inefficient memory usage
Memory leak bug	12	Specific program input, unclosed threats at runtime
Infinite loop bug	18	Specific program input, wrong API usage
Deadlock bug	13	Database binary storage, JDK related
Stuck/hang bug	17	Specific program input, configuration problem, device incompatibility
Non-performance defect	29	None

As shown in Table 4.15, there are 29 non performance defects in the analyzed sample. Filtering out the noise, there are 340-29, i.e., 311, performance defects left. Most of the false negative performance defects (251/311, i.e., 80.7%) are performance regression bugs. Performance regression bugs are inefficient or unoptimized code that may introduce performance degradations, e.g., software systems perform slowly or

use more memory or resources than previous releases [81]. However, unlike deadlock bugs or memory leak bugs, the performance regression bugs do not cause system breakdown. Thus, the functionality of the software systems can still be delivered to users with compromised performance (e.g., increased response time to answer users requests).

The main root causes of the false negative performance defects are shown in Table 4.15. We show examples of buggy files that are caused by the root causes as follows.

- Inefficient function logic: Code contains inefficient algorithms that introduce unnecessary calculations. The file, namely MekanismRenderType.java, in project Mekanism contains a performance regression bug caused by frequent unnecessary data searches (detailed description can be found in commit ee83cb14 [180]).

- Inefficient memory usage: Code holds rarely used objects in memory. The file, namely AssetFeeView.java, in project bisq contains a performance regression bug caused by building and holding a not used object (about 40MB) in memory (detailed description can be found in commit 55b070f9 [32]).

- Specific input: Performance defects that happen when specific data are input. The file, namely ColorExtractor.java, in project platform_frameworks_base contains a stuck bug that happens only when importing an image as a wallpaper (detailed description can be found in commit 5abc71b2 [12]). In addition, the file, namely DefaultSearchContext.java, in project Elasticsearch contains a memory leak bug that occurs only when a search query has certain fields enabled (detailed description can be found in commit 6b51d85c [68]).

- Wrong API usage: Performance defects that are caused by wrong API usages. The file, namely LayoutUtil.java, in project webfx contains an infinite loop bug that is caused by the mixing usage of two API methods (detailed description can be found in commit c241aa13 [260])

- Device incompatibility: Performance defects that happen because of the environment on specific devices. The file, namely BiometricsEnrollEnrolling.java, in project platform_packages_apps_settings contains a stuck bug that happens on only fingerprint devices (detailed description can be found in commit aea1bdec [13]).

- Database binary storage operations: Performance defects that are caused by wrong usages of database operations APIs. The file, namely BinaryStorageEntity.java, in project hapi-fhir contains a deadlock bug that happens when the database binary storage is used (detailed description can be found in commit c6777578 [107]).

- JDK related: Performance defects that are caused by JDK issues. The file, namely DirtyPrintStreamDecorator.java, in project buck contains a deadlock bug when the code is executed using JDK 11 (detailed description can be found in commit 2258ba13 [77]).

Based on the root causes of the false negative performance defects shown in Table 4.15, we find that there are performance defects that can only be detected using the information collected from running time (e.g., testing phase), such as performance defects caused by specific input, device incompatibility, and JDK issues. For the performance defects caused by other root causes (i.e., inefficient function logic, inefficient

memory usage, wrong API usage, and database binary storage operations), the code characteristics of the root causes can be system-specific as different systems have different coding practices. In the future, we plan to propose system-specific performance code metrics to measure the code characteristics of inefficient function logic, inefficient memory usage, wrong API usage, and database binary storage operations in software systems.

4.5.3 Usage of our approach

In practice, developers may not have sufficient time to design performance test cases to cover each file or function in a software system [55]. To this end, developers can use our approach to prioritize their testing effort on a small set of source code files that are predicted to have performance defects and improve the performance test coverage of software. As the first step, developers can use our approach to predict files that have performance defects. Then, they can introduce performance test cases to test the performance (e.g., execution time, CPU usages, and memory usages) of files that are considered to have performance defects. In addition, developers can use approaches proposed in the existing studies [44, 62] to test the performance defects. Chen et al. [44] propose an approach to predict which test cases are likely to manifest performance regression in a commit and Ding et al. [62] propose an approach to predict whether a function test case is able to demonstrate the performance improvement after fixing a performance defect. Developers can use the approach proposed by Chen et al. [44] to select the existing test cases that can manifest the performance regression in the files. After fixing the performance defects, developers can use the approach proposed by Ding et al. [62] to identify the test cases that can demonstrate

the performance improvement.

Our approach uses performance code metrics, code metrics, and process metrics as predictors to predict performance defects based on the current snapshot of the code. When developers modify the source code to fix the predicted performance defects, the values of the metrics are changed. Thus, developers should not re-run our approach to verify the fixes for performance defects. Instead, developers need to run test cases to test the modified files to ensure the performance defects are fixed.

4.5.4 Predicting performance bugs at a fine grained level

Our approach identifies the files that contain performance bugs, but our approach does not provide information (e.g., exact lines of code where the bugs are in the file) to guide developers to fix the performance bugs. We evaluate the performance of machine learning algorithms for predicting performance bugs at method level instead of file level. Our findings suggest that the best median MCC, PR-AUC, and AUC achieved by the machine learning algorithms are 0.06, 0.01, and 0.59, respectively, for predicting methods that have performance bugs. Also, we find that there is only a median of 0.17% methods that have performance bugs in our projects. The percentage of methods that have performance bugs is too low and it is challenging to train the machine learning models to predict performance buggy methods.

4.5.5 Conducting cross-project performance bugs prediction

.

We test the performance of machine learning algorithms for predicting performance bugs under cross-project settings, i.e., training machine learning algorithms

using data from one project and predicting performance bugs in another project. We find that the prediction models achieve a median of 0.54 AUC, 0.06 PR-AUC, and 0.11 MCC for predicting performance bugs under cross-project settings. One possible reason behind the poor performance is that the history performance bugs from one project cannot represent the performance bugs in other projects because projects are from different application domains and have various coding styles.

4.6 Threats to Validity

In this section, we discuss the threats to the validity of experiments conducted in this chapter.

Threats to external validity are related to the generalizability of our results. To ensure the generalizability of our approach, we conduct experiments on 80 popular Java projects. These projects belong to different domains. However, testing our approach on only GitHub open-source projects can introduce generalizability concerns about our conclusions. Studying software systems that are programmed in other languages and industry software systems can be useful to augment the generalizability of our approach.

We study the characteristics of the performance defects and non-performance defects from 80 GitHub projects. However, we do not investigate the performance defects and non-performance defects from other issue tracking systems such as Bugzilla [37]. To augment the generalizability of our findings, we have to reproduce our work in different project settings, such as projects from other open-source platforms and corporate software projects.

A limitation of our approach is to predict performance defects in new project with

limited development history. This is because our approach relies on the performance defects history of a project to train machine learning models in predicting hidden performance defects in the same project.

We suggest metrics to represent the code characteristics of the performance defects. However, we have not included every known type of performance defects in this chapter. To augment the generalizability of using machine learning algorithms to predict performance defects, we have to propose metrics from more types of performance defects.

Threats to internal validity concern factors out of our control that may affect the experiment results. One internal threat to our results is that we use heuristics to categorize the artifacts in Java source code. Developers in different projects tend to define their own classes (e.g., file classes) to fit their needs. It is challenging to manually go through each project, collect the classes, and classify the classes into different categories. Thus, we classify the classes via keyword searching. For example, if the class of an object contains the keyword *file*, then the corresponding object is classified as a File object and the operations on the object are considered as file operations. However, it is impossible to avoid having noise in our heuristic approach (i.e., misclassified classes).

Similarly, we use keyword searching to identify the performance defect issue reports and the performance defect fixing commits following the prior studies. However, there might be *false positive* performance defect issue reports and performance defect fixing commits in our dataset. To mitigate this issue, we manually labelled a random statistical sample to ensure that our results does not have many false positives.

We use performance defect fixing commits to identify source code files that have

performance bugs. However, a performance bug fix might spread over several commits and one commit can implement multiple unrelated changes (e.g., feature enhancement and performance bug fixing). Thus, there might be false positive performance bug files collected in our experiment dataset.

4.7 Summary

In this chapter, we present an approach that predicts performance defects in software systems by integrating source code analysis and mining code change history. We propose performance code metrics to capture the code characteristics of performance defects. Models are built to predict performance defects in software systems using code, process, and performance code metrics. We conduct extensive experiments on 80 Java projects to evaluate the performance of seven machine learning algorithms for predicting performance defects.

Our results suggest:

- The performance defects are more time-consuming to fix than the non-performance defects as fixing a performance defect takes twice more developers and 32.3% more time than fixing a non-performance defect.
- The Random Forest and eXtreme Gradient Boosting algorithms achieve the best performance in predicting files that have performance defects with a median of 0.38 MCC, 0.21 PR-AUC, and 0.84 AUC.
- The proposed performance code metrics are the most important metrics in predicting performance defects. We find that the AUC, PR-AUC, and MCC of the studied machine learning models drop a median of 7.7%, 25.4%, and 20.2%

without using the proposed performance code metrics.

Chapter 5

An Empirical Study On Predicting Buggy Methods And Their Fixing Effort

To help developers identify defects and allocate limited testing resources, existing studies propose approaches to analyze the descriptions and attributes (e.g., reporters, severity labels, and affected software components) of new issue reports and estimate the effort for fixing the defects described in the issue reports. However, existing studies require information from issue reports to predict the fixing effort of defects. Therefore, the existing studies cannot provide estimation about the fixing effort for the predicted defects which have not been reported by developers. Existing defect prediction approaches can predict hundreds of defects in a software. There are defects that can be solved quickly (i.e., within a day), while some defects can take more than a month to fix. In this chapter, we present an approach that predicts the fixing effort categories of the predicted buggy methods.

Chapter organization: Section 5.1 describes the introduction of this chapter. Section 5.2 outlines the design of our approach and the data collection method. Section 5.3 presents the evaluation results for our approach. The threats to validity

are discussed in Section 5.4, Section 5.5 summarizes the chapter and discusses future work.

5.1 Introduction

The increasing scale and complexity of software systems have driven software practitioners to seek new solutions to reduce the number of defects in order to increase the reliability of software. As mentioned by Liu et al. [170], the defects in source code are the main causes of failures, which dramatically impact the reliability of software systems. Due to the time-to-market, it is challenging for developers to manually localize all the defects before software releases [153]. Various defect prediction approaches have been proposed to automatically identify potential defects and help developers allocate their limited resources [108, 122, 167, 255, 256, 281].

Generally speaking, defect prediction approaches [54, 78, 108, 167, 168, 255, 256, 281] mainly consist of two steps: (1) training machine learning or deep learning models using source code history (e.g, commits) of software systems; (2) and using the trained models to predict whether new code artifacts (e.g., files, methods, or commits) contain defects. After predicting the code artifacts that have defects (e.g., methods that have defects) in software systems, developers investigate the predicted defects in order to address them. As mentioned in existing studies [28, 118, 160], the estimation of fixing effort (i.e., time needed to fix a defect) is critical to help developers make decisions in coordinating effort during bug triaging, resource allocation for fixing defects, and release management. Figure 5.1 shows the total time taken to fix defects in methods in 106 open-source Java projects. As shown in Figure 5.1, 25% buggy methods can be fixed within a day, while more than 50% buggy methods need more than a week to

be fixed. The estimation of the effort needed to fix buggy methods enables developers to select buggy methods to fix based on their working schedule. However, existing defect prediction studies provide no information about the estimated fixing effort for the predicted defects.

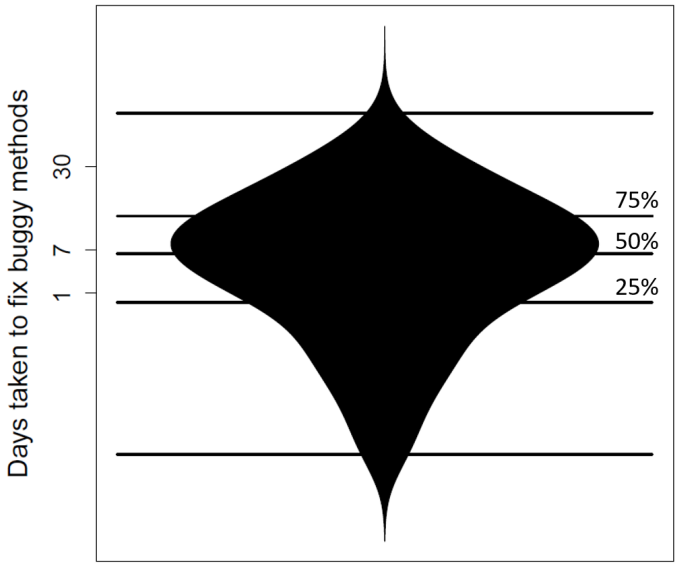


Figure 5.1: Time taken to solve buggy methods for the studied 106 Java projects

Various approaches [8, 28, 88, 118, 160, 176, 261, 285] have been proposed to predict fixing effort for defects. The existing approaches analyze the descriptions and attributes (e.g., reporters, severity labels, and the affected software components) of a new issue report and estimate the effort for fixing the defect described in the issue report. For example, Weiss et al. [261] predict the fixing effort of a new issue report by analyzing the resolved issue reports with similar textual descriptions. The above studies require information from issue reports to predict the effort for fixing a defect. However, the issue reports are not available for the predicted defects as the predicted defects have not been reported by developers. Therefore, the above approaches cannot

be used for the predicted defects. In this chapter, we strive for predicting fixing effort for the predicted defects to assist developers in allocating limited resources to meet the pressing software release deadline.

More specifically, we predict buggy methods, i.e., methods that have any types of defects, using CodeBERT [80], a pre-trained model that captures the semantic connection between the comments of a method and the source code [80]. We choose CodeBERT because only the source code of methods is required to fine tune CodeBERT, while other approaches [54, 78, 108, 167, 168, 255, 256, 281] need to calculate metrics or build Abstract Syntax Trees (ASTs) to represent the source code of methods before predicting defects. To provide an estimation of fixing effort for the predicted buggy methods, we train models using six traditional machine learning algorithms, such as Random Forest, eXtreme Gradient Boosting, and Support Vector Machines. Specifically, we categorize buggy methods into three fixing effort categories based on the time spent on fixing the defects by developers as follows:

- Low fixing effort: buggy methods that can be fixed within a day
- Medium fixing effort: buggy methods that need more than a day but less than a week to be fixed
- High fixing effort: buggy methods that need more than a week to be fixed

To evaluate our approach, we conduct experiments on 106 open-source Java projects from GitHub. Our work aims to address the following research questions (RQs):

RQ1. What is the performance of our approach for predicting the fixing effort categories of the predicted buggy methods?

We use code metrics, process metrics, and semantic metrics to build prediction models to estimate the fixing effort categories. We test the performance of six widely-used multi-class classification machine learning algorithms (i.e., Complement Naive Bayes, K-Nearest Neighbors, MultiLayer Perceptron, Support Vector Machine, Random Forest, and eXtreme Gradient Boosting) for predicting the fixing effort categories of the predicted buggy methods. Our results suggest that the eXtreme Gradient Boosting (XGBoost) algorithm achieves the best performance for predicting fixing effort categories of time period with a median of 0.5 MCC, 0.8 of Weighted Precision, and 0.8 of Weighted Recall.

RQ2. What are the most significant groups of metrics that affect the performance of our models?

To understand which groups of metrics are useful in predicting the fixing effort of the predicted buggy methods, we analyze the effects that each group of metrics has on the fixing effort of our models. We calculate the effect of metrics per group because the semantic metrics group has a high number of metrics (e.g., 300 semantic metrics in project Elasticsearch [75]) and it is time-consuming to measure the effect of each metric in the semantic metrics group. We find that the semantic metrics are the most important metrics in predicting the fixing effort of the predicted buggy methods. Without using the semantic metrics, the MCC, Weighted Precision, and Weighted Recall drop an average of 27.4%, 16.7%, and 11.9% for the studied machine learning algorithms, respectively.

5.2 Experiment Setup

In this section, we introduce the overview of our experimental setup. As shown in Figure 5.2, we first collect data from GitHub and, then, identify buggy methods which we label their fixing effort categories. Then, we apply CodeBERT to predict buggy methods and build machine learning models to predict the fixing effort categories of the predicted buggy methods.

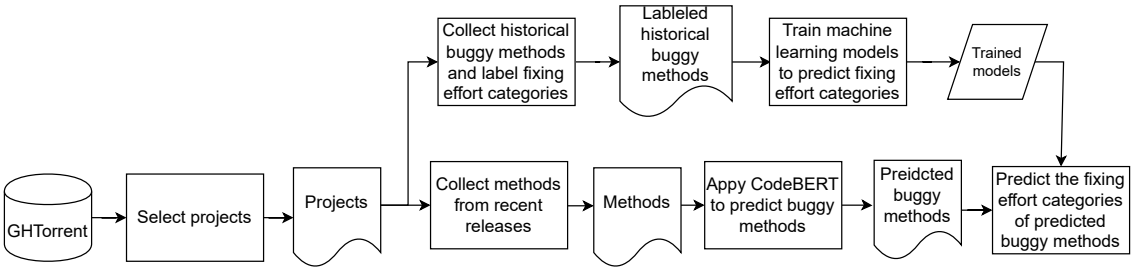


Figure 5.2: The overview of our approach.

5.2.1 Projects selection

As shown in Figure 5.2, we first select an initial set of experiment projects after querying GHTorrent's dataset [94] . GHTorrent is a widely used dataset that hosts data collected from GitHub projects [95, 97, 274], such as creation times of projects, the numbers of commits in projects, and the numbers of developers in projects.

We use a set of criteria to select Java projects for this chapter. Java is one of the most popular programming languages used in both industrial and open-source projects according to programming language popularity websites, e.g., Tiobe [243], GitHut [91], and PYPL [210]. Although we have used only Java projects, our approach can predict fixing effort for buggy methods written in other programming languages, assuming that the metrics of buggy methods are properly calculated. To

avoid working on personal or toy projects [143], we first select 1,697 Java projects that have at least 2,000 commits [221] from GHTorrent's dataset. Next, we apply the following criteria to further exclude projects that do not have enough development history to evaluate our defect fixing effort prediction approach.

- As mentioned by Gousios et al. [93], the majority of the GitHub projects are forked from other projects. To avoid working on inactive projects, we exclude projects that are forks or have been archived [89].

- We remove projects that have less than a year of development history. To obtain enough history data for building and testing our fixing effort prediction approach, we require the experiment projects to have at least a year of development history.

- We further exclude projects that do not have defect fixing commits in the latest six months when we started collecting our proejcts [283]. Following the prior study [283], we use the defect fixing commits in the latest six months to identify buggy methods from recent releases. Thus, projects with no defect fixing commits in the latest six months are removed from our dataset.

- We filter out projects that do not manage issue reports on GitHub. We use the issue reports to identify the fixing effort of buggy methods (i.e., time spent on fixing the buggy methods). We include the projects that have issue reports on GitHub as we use issue reports to label the fixing effort of buggy methods. However, our approach can be adapted to projects that use external defect tracking systems (e.g., Jira and Bugzilla), assuming that the issue reports can be automatically downloaded and parsed.

- We further exclude projects that have few issue reports otherwise we can not train reliable models with high predictive accuracy. Following the prior studies [282, 283], we use the 75% percentile of the number of issue reports (i.e., 1,418 issue reports in our experiment) as the threshold to filter out projects.

After applying the aforementioned criteria, the number of experiment projects is reduced to 106. The selected projects cover a range of different domains, such as Elasticsearch search engine [75] and OpenLiberty cloud micro-service [125]. On average, the collected projects have $3,740$ source code files and $437,433$ lines of code.

5.2.2 Collecting methods from the recent releases of projects

As shown in Figure 5.2, we collect methods from recent releases of projects and fine tune CodeBERT models to predict the buggy methods of our projects. As suggested in prior studies [282, 283], we collect the source code files of each project six months before the latest commit time for each project. For example, in project Elasticsearch, if the latest commit time is Dec. 7, 2021 and we obtain the source code files from the project snapshot on Jun. 7, 2020 (i.e., 6 months before Dec 7, 2021). Then, we identify the defect fixing commits in the six months before the latest commit time and locate the files that are modified in these defect fixing commits to fix the defects. We analyze the modified files and label the buggy methods that are changed in the defect fixing commits. We exclude the defects introduced after the time we collect the source code files. Figure 5.3 below illustrates each step in details.

Identifying historical defect fixing commits. To identify the defect fixing commits, we search for commits that contain any of the defect fixing related keywords in the commit message (i.e., *defect*, *bug*, *fix*, and *patch*), as suggested in prior

```
commit 2d3df921bcdb30ae607048d96650625a2ad2a904
Author: Martijn van Groningen <martijn.v.groningen@gmail.com>
Date:   Thu Nov 28 17:21:29 2013 +0100

    Fixed positive infinity bug that can occur in specific scenarios
 when score mode average is used.

    Closes #4291
```

(a) Defect fixing commit 2d3df [249] in project Elasticsearch

```
@Override
public int advance(int target) throws IOException {
    currentDocId = parentsIterator.advance(target);
    if (currentDocId == DocIdSetIterator.NO_MORE_DOCS) {
        return currentDocId;
    }
```

(b) Buggy method advance in file ChildrenQuery.java in project Elasticsearch

```
@Override
public int advance(int target) throws IOException {
    if (remaining == 0) {
        currentDocId = NO_MORE_DOCS;
        return NO_MORE_DOCS;
    }
```

(c) Method advance in file ChildrenQuery.java after fix from commit 2d3df [249]

Figure 5.3: Examples of a defect fixing commit and the modified method in the defect fixing commit

studies [144, 150, 282]. For example, commit *2d3df* [249], in Elasticsearch project, contains the keyword *bug* and is identified as a defect fixing commit by our keyword-based approach as shown in Figure 5.3(a). Moreover, we keep only commits that contain source code changes to Java files. As shown in Figure 5.4, there is a median of 119 defect fixing commits per project.

Identifying files that are modified in the defect fixing commits. We query the commit information stored in the GitHub repositories to identify the files modified in the defect fixing commits. For example, for defect fixing commit *2d3df* [249] shown in Figure 5.3(a), we identify the modified file *ChildrenQuery.java* [248] by querying

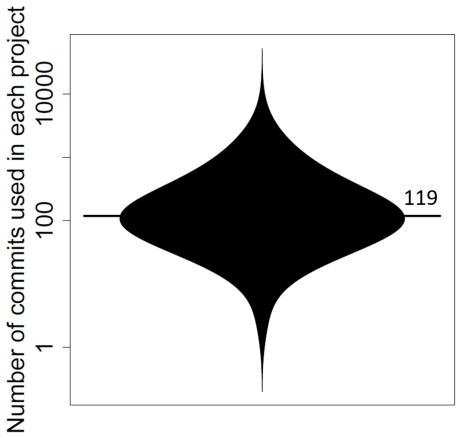


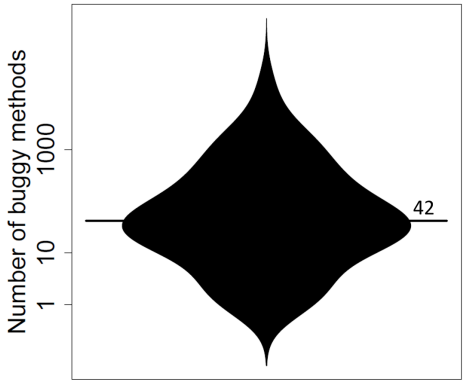
Figure 5.4: The number of commits that are collected from each experiment project.

the GitHub repository of the Elasticsearch project. Test files that are modified in the defect fixing commits are excluded since we are interested in studying the effort of developers in fixing defects that affect the behaviors of software systems.

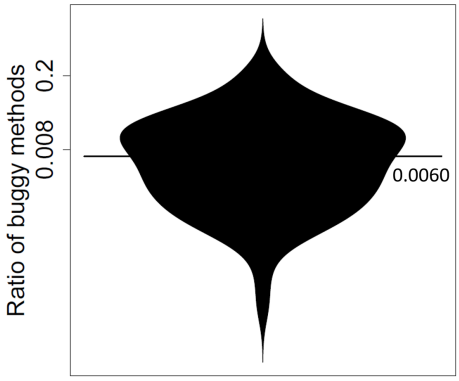
Labeling historical buggy methods that are modified in defect fixing commits. After identifying the modified files from the defect fixing commits, we locate the methods that are modified in these files. We consider the modified methods in the defect fixing commits as buggy. To obtain changed methods in a file, we use the FinerGit tool [115]. FinerGit can fetch the changed Java methods in a committed file by comparing tokens of methods before and after a commit. The methods that have different tokens after a commit are considered as modified [115]. As shown in Figures 5.3(b) and 5.3(c), the method *advance* is changed in file *ChildrenQuery.java* in commit $2d3df$ [249]. Therefore, we consider *advance* as a buggy method for the corresponding project.

Excluding noisy buggy methods. We are aware that there might be defects introduced after the time we collect the methods (i.e., introduced within the latest six months) in a project. Thus, we use SZZ [222] algorithm to search for the defect

introducing commits. We exclude the buggy methods that are introduced after the time that we collect the source code methods. Figure 5.5 shows the number and the ratio of the buggy methods in the 106 Java projects. In our projects, there is a median number and ratio of 42 and 0.60% buggy methods, respectively.



(a) The number of the buggy methods in each project



(b) The ratio of the buggy methods in each project

Figure 5.5: The number and the ratio of the buggy methods in the 106 GitHub projects for fine tuning and evaluating CodeBERT models

5.2.3 Applying CodeBERT to predict buggy methods

As shown in Figure 5.2, after collecting buggy and clean methods from the selected projects, we fine tune CodeBERT models to predict buggy methods.

Pipeline of using CodeBERT. The pipeline of using CodeBERT to predict buggy methods is shown in Figure 5.6. Following prior study [172], we attach a `feedforward` neural network (FFNN) with a `softmax` layer to the pre-trained CodeBERT model. The added `feedforward` neural network (FFNN) serves as an output layer to convert the semantic metrics of a method output by CodeBERT to the probability of the method being buggy. If the probability is greater than 0.5, the method is

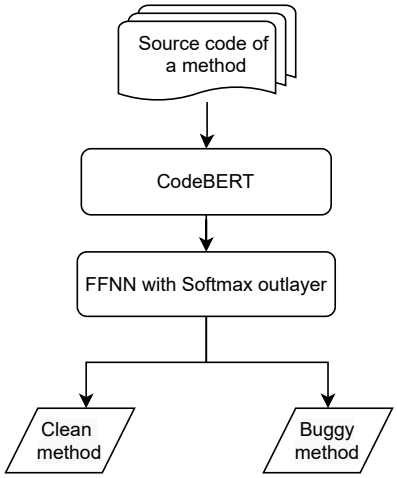


Figure 5.6: The pipeline of using CodeBERT to predict buggy methods.

considered to be buggy. Otherwise, the method is predicted to be a clean method. We use the open-source implementation[1] of the pipeline made available by Microsoft [172]. Table 5.1 lists the optimal hyper-parameters for the CodeBERT models. The detailed descriptions of the parameters can be found in the online documentation provided by Microsoft.[1]

Table 5.1: The parameters that are used to fine tune CodeBERT models

Parameter	**Value**
Block size	400
Training batch size	32
Evaluating batch size	64
Number of epoches	5
Learning rate	2e-5
Max gradient norm	1.0

Out-of-sample bootstrap evaluation approach. In this chapter, we fine tune CodeBERT models to conduct within-project buggy method prediction, i.e., a CodeBERT model is fine tuned and used to predict buggy methods on the same

[1]https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/Defect-detection

project [282]. We follow the out-of-sample bootstrap evaluation approach [65] to fine tune CodeBERT models and apply CodeBERT models to predict buggy methods. Out-of-sample bootstrap evaluation approach has been widely used in prior studies [5, 65, 237, 238] to obtain less biased and more stable prediction results than the k-fold cross-validation for skewed datasets, which is the case for ours (see Figure 5.5). The out-of-sample bootstrap consists of two steps:

- For a project with N methods, we randomly select the same number of methods (i.e., N) with replacement from the original methods of the project. The randomly selected methods are referred to as bootstrap sample.
- The bootstrap sample is used to fine tune and validate the pre-trained CodeBERT model to predict buggy methods. Then, the fine tuned CodeBERT model is tested using the methods that do not appear in the bootstrap sample. Since the bootstrap sample is selected with replacement, 36.8% of the methods will not appear in the bootstrap sample [65].

For each project, we repeat the above steps 100 times to obtain a robust prediction results of CodeBERT models for predicting buggy methods. In each of the 100 iterations, we collect the methods that are predicted to be buggy by the CodeBERT model and exclude the false positives, i.e., clean methods that are predicted as buggy methods. As shown in Figure 5.2, the predicted buggy methods are then used to evaluate our fixing effort prediction approach. Through our experiments, we observe that the CodeBERT achieves a median of 0.90 AUC, 0.61 MCC, and 0.58 PR-AUC in predicting buggy methods in the experiment projects as shown in Figure 5.7.

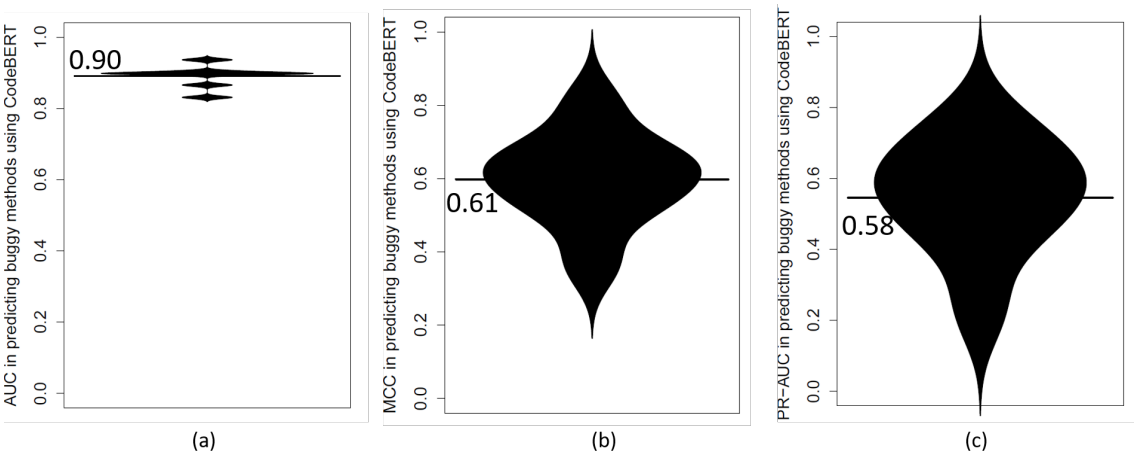


Figure 5.7: The performance of CodeBERT for predicting buggy methods for the 106 Java projects.

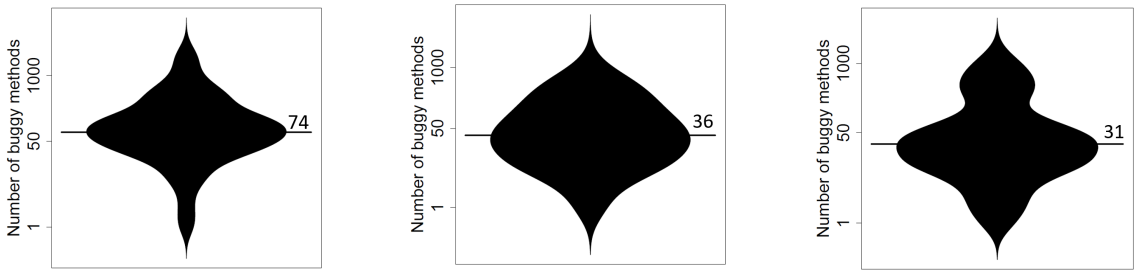
5.2.4 Collecting historical buggy methods and label fixing effort categories

Collect historical buggy methods. As shown in Figure 5.2, we use the historical buggy methods to build machine learning models for predicting the fixing effort categories of buggy methods. We first collect defect fixing commits before the latest six month periods, and then we collect historical buggy methods that are modified in the collected defect fixing commits.

Label the fixing effort categories of buggy methods. We label the historical buggy methods and the predicted buggy methods with fixing effort using the time intervals from when a developer starts addressing a buggy method to when the developer finishes addressing the buggy method. The commit time of a buggy method can be considered as the time a developer finishes addressing the corresponding buggy method. The commit information in GitHub repositories does not contain the time when a developer starts modifying the source code. Thus, it is challenging to obtain

the exact time a developer starts working to fix a buggy method. Usually, in GitHub projects, a developer includes the issue report ID in a commit message that the associated commit addresses. For example, a developer records issue report ID 4291 in commit $2d3df$ [249] as shown in Figure 5.3(a). After a issue report is assigned to one or several developers, the assigned developers start fixing the defect [130]. Therefore, we consider the assignment time of a issue report as the time a developer starts addressing the defect.

For each defect fixing commit, we search for the issue report ID in its commit message and find the assignment time of the issue report. We use regular expressions to search for the issue report ID in the commit messages. If the IDs of multiple issue reports are identified, we use the latest issue report (i.e., the issue report with the latest submission time). For a defect fixing commit that we cannot find any issue report IDs in the commit messages, we exclude the buggy methods collected from the defect fixing commits.



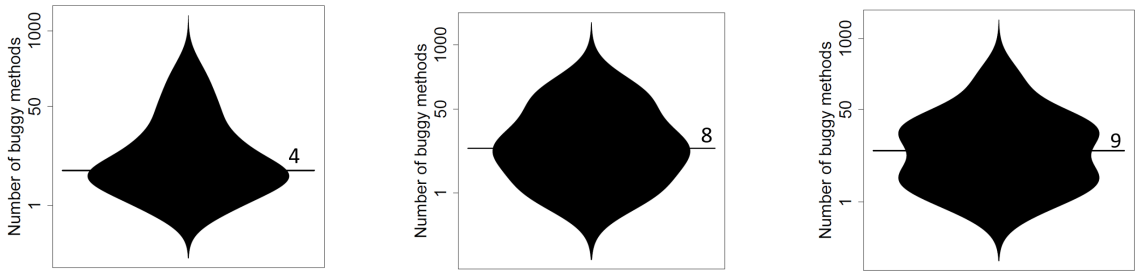
(a) The number of training buggy methods in low fixing effort category in each project

(b) The number of training buggy methods in medium fixing effort category in each project

(c) The number of training buggy methods in high fixing effort category in each project

Figure 5.8: The number of buggy methods in training datasets of the 106 GitHub projects

Training and testing datasets for our fixing effort prediction approach.



(a) The number of testing buggy methods in low fixing effort category in each project

(b) The number of testing buggy methods in medium fixing effort category in each project

(c) The number of testing buggy methods in high fixing effort category in each project

Figure 5.9: The number of buggy methods in testing datasets of the 106 GitHub projects

For each project, the historical buggy methods before the latest six months periods are collected as the training dataset. Figure 5.8 shows the number of collected historical buggy methods with fixing effort categories for the 106 Java projects. In particular, for our projects, there is a median number of 74, 36, and 31 buggy methods in the low, medium, and high fixing effort categories, respectively.

The predicted buggy methods using CodeBERT models are served as the testing dataset. Figure 5.9 shows the number of buggy methods that are predicted by CodeBERT models in each fixing effort category in the 106 Java projects.

5.3 Results and Analysis

In this section, we present the motivations, approaches, and results of the studied research questions.

5.3.1 RQ1. What is the performance of our approach for predicting the fixing effort categories of buggy methods?

Motivation: Prior defect prediction studies [54, 108, 122, 167, 255, 256, 281] do not provide information about the estimated effort for fixing the predicted defects. As mentioned in the existing studies [28, 118, 160], predicting fixing effort for defects can help project teams balance developers' workload and achieve better prioritization. However, the existing studies require information from issue reports (e.g., reporters and textual description about defects) to train a defect effort prediction model. Therefore, estimating the fixing effort for the predicted buggy methods is not feasible using the existing approaches as the issue reports are not available for the predicted buggy methods. To address such limitation, in this RQ, we propose an approach for predicting fixing effort categories of the predicted buggy methods.

Approach. We calculate metrics as predictors for building machine learning models and evaluate their performance in predicting the fixing effort categories of the predicted buggy methods in the following steps:

Calculating metrics as predictors for predicting defect fixing effort of the predicted buggy methods. To build the prediction models, we use the following three categories of metrics, i.e., code metrics, process metrics, and semantic metrics. The metrics used in this chapter are shown in Table 5.2.

Code metrics. Measure source code characteristics of each method. For each buggy method in our datasets, we calculate the code metrics by analyzing the collected source code of the buggy methods. To do that, we use the Understand tool [1] which is a well-established static analyzer that collects source code metrics and it has been used in prior studies [5, 281, 283].

Process metrics. Measure code change history of each method. For each buggy method in our datasets, the process metrics are calculated using the code change history before the time a buggy method was introduced. We use SZZ [222] algorithm to point out the time that the buggy methods were introduced. We use the aforementioned FinerGit tool [115] to obtain the change histories of Java methods. Then, we implement scripts to automatically query the code change history (i.e., commits) of a method and calculate the process metrics.

Semantic metrics. Capture the semantic meaning of each method. For each method, we use the directory, filename, method name, and comments to encode both module information and code behavioral information of the buggy method [150, 220]. We first tokenize the directories of buggy methods and we apply the camel case split to further split the filenames and method names. Next, we perform word stemming to obtain the stems of words obtained from the directories, filenames and method names. Then, we apply the Bag of word (BOW) approach [217] to extract metrics and represent the textual information of the buggy methods.

Selecting machine learning algorithms. Because there are three categories (i.e., low, medium, and high fixing effort group), we select machine learning algorithms that can perform multi-class predictions. Specifically, we select six widely used machine learning algorithms, i.e., Random Forest (RF), eXtreme Gradient Boosting (XGB), Support Vector Machines (SVM), Complement Naive Bayes (CNB), Multi-Layer Perceptron neural network (MLP), and K-Nearest Neighbors (KNN).

Applying PCA to reduce the dimensionality of semantic metrics. We use semantic metrics to capture both module information and code behavioral information of the buggy methods. However, the number of semantic metrics can be quite large.

Table 5.2: The metrics used in this chapter.

Metric type	**Metric name**	**Description**
Code metrics	LOC	Lines of code in a method
	CL	Comment lines in a method
	NSTMT	Number of statements in a method
	RCC	Ratio comments to codes of a method
	MNL	Max nesting level of a method
	CC	McCable cyclomatic complexity of a method
	FANIN	Number of input data of a method
	FANOUT	Number of output data of a method
Process metrics	Num_rev	Number of times a method was changed
	Num_fixing_rev	Number of times of method was changed to fix defects
	Num_authors	Number of distinct authors that change a method
	Added_loc	Number of lines added to a method body in the history commits
	Deleted_loc	Number of lines deleted from a method body in the history commits
Semantic metrics	semantic_metrics	Convert the directory, filename, method name, and comments of a method into features using BOW technique.

On average, we extract 205 semantic metrics from each of the 106 projects. Feeding a high number of semantic metrics directly to a machine learning algorithm can lead to overfitting [230]. Similarly to prior studies [21, 60, 119, 126, 165], we apply principal component analysis (PCA) [268] to reduce the dimensionality of semantic metrics.

PCA first creates new uncorrelated variables from the semantic metrics of a buggy method. The generated variables contain all the information contained in the input semantic metrics. To reduce the dimensionality of the semantic metrics, we need to set up a variance ratio (i.e., the percentage of information we intend to keep from the semantic metrics). Then, PCA reduces the number of generated variables and returns a minimum number of variables that can cover the specified variance ratio. Next, we use the generated variables, returned by PCA, to represent the semantic metrics of the buggy methods.

Performance metrics. We apply three widely used performance metrics, i.e., Matthews correlation coefficient (MCC), weighted Precision, and weighted Recall to measure the performance of our fixing effort prediction models. For a buggy method, the machine learning models are trained to output the probabilities of the buggy method for being in each of the three fixing effort categories. Then, the category with the highest probability is then considered as the predicted fixing effort category for the buggy methods. If the predicted fixing effort category for a buggy method is the same as the real fixing effort category of the buggy method, we consider that a model makes a *true positive* predictions. Otherwise, the models make a *false positive* prediction. Precision measures the percentage of *true positive* predictions among all the predictions made by machine learning models. Recall represents the percentage of buggy methods in a fixing effort category that can be predicted by a machine learning model among all the buggy methods in the fixing effort category in the datasets.

We combine the Precision and Recall for each fixing effort category and compute the weight Precision and weighted Recall to measure the overall performance of our

models for predicting fixing effort categories of the collected buggy methods. Equations 5.1 and 5.2 show the computation for weighted precision and weighted recall for each fixing effort category.

$$Weighted\_precision = \frac{P_{low} * N_{low} + P_{medium} * N_{medium} + P_{high} * N_{high}}{N_{low} + N_{medium} + N_{high}} \quad (5.1)$$

$$Weighted\_recall = \frac{R_{low} * N_{low} + R_{medium} * N_{medium} + R_{high} * N_{high}}{N_{low} + N_{medium} + N_{high}} \quad (5.2)$$

Where N_{low}, N_{medium}, and N_{high} are the number of buggy methods for the low, medium, and high fixing effort groups, respectively. P_{low}, P_{medium}, and P_{high} are the Precision of models for predicting buggy methods as low, medium, and high fixing effort categories. Similarly, R_{low}, R_{medium}, and R_{high} are the Recall of models.

Training machine learning models. To find the best hyper-parameters for our machine learning models and the variance ratio for PCA on each experiment project, we apply the grid search optimization approach [238] mentioned in Section 4.4 of Chapter 4.

Evaluating the performance of prediction models. After selecting the optimal hyper-parameters for our machine learning models on each project, we evaluate the performance of our models on the testing datasets of the project. For each project, we train a machine learning model on the training dataset using the optimal hyper-parameters and test the MCC, weighted Precision, and weighted Recall for predicting the fixing effort categories of the predicted buggy methods in the testing dataset. Because we have six machine learning algorithms and 106 projects, we build and test 636 (i.e., 6×106) models, in total.

Comparing the performance of the models. After evaluating the performance of six machine learning models on 106 open-source projects, we draw beanplots of the MCCs, weighted Precision, and weighted Recall to visualize the performance of our models. Next, we conduct Scott-Knott ESD test [129] to compare the performance of different models. As suggested in the prior studies [238, 284], the Scott-Knott ESD test can overcome the overlapping comparisons that happen in other tests, such as the Mann-Whitney U test [219] and Nemenyi's test [190].

For each of the performance metrics (i.e., MCC, weighted precision, and weighted recall), we apply the Scott-Knott EST test to compare the performance of the studied machine learning algorithms. For a performance metric, the Scott-Knott EST test ranks the machine learning algorithms into different groups where the algorithms in different groups have significantly different performance metrics. The Scott-Knott ESD test recursively ranks the machine learning algorithms through hierarchical clustering analysis. In each iteration, the Scott-Knott ESD test separates the algorithms into two groups based on the performance metric. If the two groups have significantly different values of the performance metric, the Scott-Knott ESD test executes again within each group. If no statistically distinct groups can be created, the Scott-Knott ESD test stops [87]. Also, we set up a 95% confidence level to test if two groups are significantly different when applying the Scott-Knott ESD test.

Results: The eXtreme Gradient Boosting algorithm achieves the best performance in predicting the fixing effort categories of the predicted buggy methods. As shown in Figures 5.10, 5.11, and 5.12, the eXtreme Gradient Boosting algorithm achieves a median of 0.5 MCC, 0.81 weighted Precision, and 0.8 weighted Recall. The colors in Figures 5.10, 5.11, and 5.12 represent the distinct

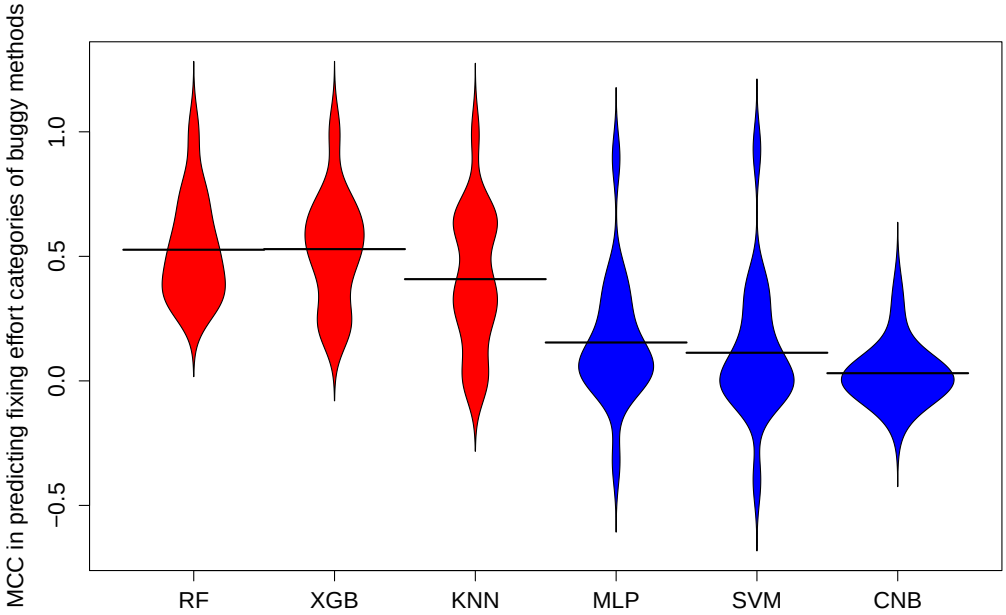


Figure 5.10: The MCC of machine learning algorithms for predicting the fixing effort categories of the predicted buggy methods

groups of machine learning algorithms ranked by the Scott-Knott ESD tests. The Scott-Knott ESD tests always rank the K-Nearest Neighbors, Random Forest, and eXtreme Gradient Boosting algorithms into the high performance group when testing on MCCs, weighted Precision, and weighted Recall. As shown in Figures 5.10, 5.11, and 5.12, the Complement Naive Bayes and Support Vector Machines algorithms have the worst performance for predicting the fixing effort categories of predicted buggy methods with medians of MCCs close to 0, which indicates more or less a random guessing for the fixing effort categories. A possible reason is that the Complement Naive Bayes and Support Vector Machines algorithms are not suitable for the multi-class prediction problem.

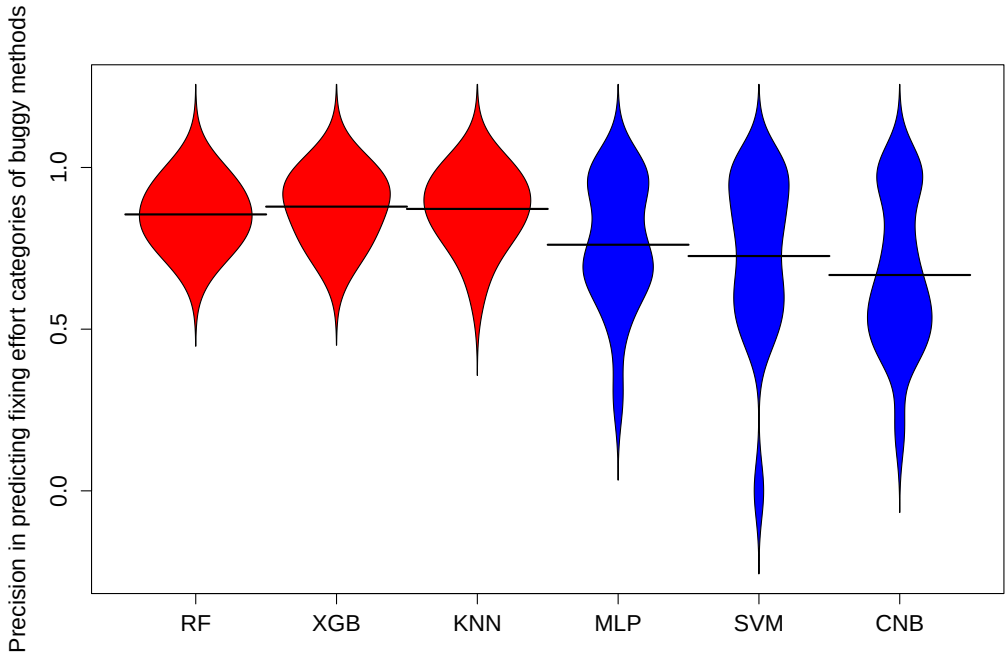


Figure 5.11: The precision of machine learning algorithms for predicting the fixing effort categories of the predicted buggy methods

Summary

The eXtreme Gradient Boosting (XGB) algorithm achieves the best performance in predicting the fixing effort categories of the predicted buggy methods with a median of 0.5 MCC, 0.81 weighted Precision, and 0.8 weighted Recall.

5.3.2 RQ2. What are the most significant groups of metrics that affect the prediction performance of our models?

Motivation: To build fixing effort prediction models, we use three groups of metrics including code metrics, process metrics, and semantic metrics. To understand which groups of metrics are the most useful in predicting the fixing effort of the predicted

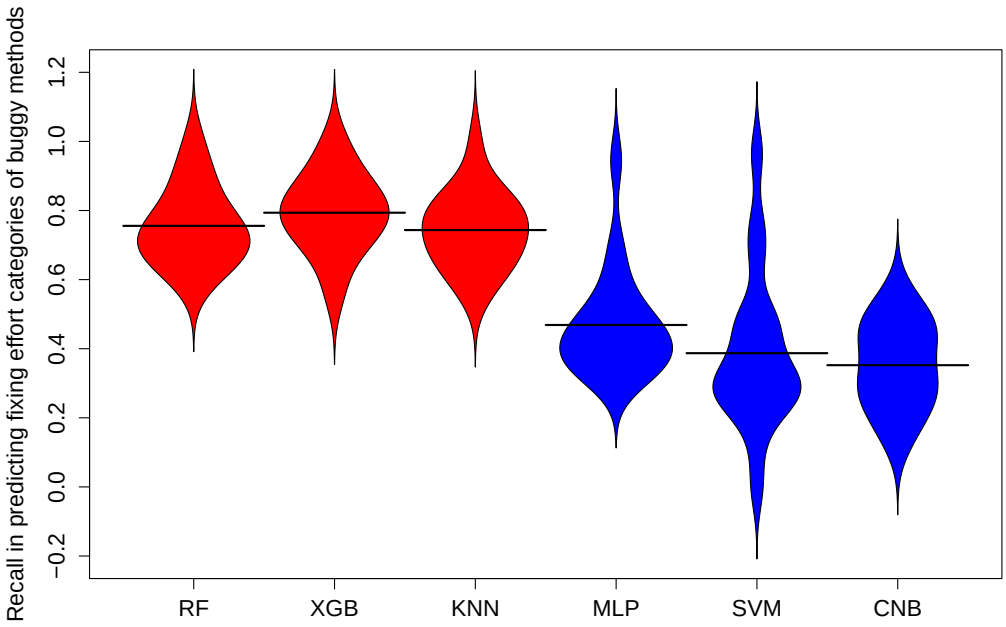


Figure 5.12: The recall of machine learning algorithms for predicting the fixing effort categories of the predicted buggy methods

buggy methods, we analyze the effects that each group of metrics has on our prediction models. As mentioned in Section 5.3.1, the experiment projects have an average of 205 semantic metrics. Due to the high number of semantic metrics, we do not calculate the effect of each metric, instead we measure the effects of groups of metrics.

Approach: To test the effect of a group of metrics on our machine learning models, we apply the effect calculation process as suggested in prior study [238]. More specifically, we use the following two steps:

- For the training and testing datasets from a project, we first randomly permute all the values of a group of metrics and obtain a new dataset.

- We apply out-of-sample bootstrap process on the new training dataset and

conduct the parameter optimization to select the optimal hyper-parameters for our machine learning algorithms. Next, we evaluate the performance of the machine learning algorithms for predicting the fixing effort categories of the predicted buggy methods on the new testing dataset. We compute the differences between the performance of the models that are built on the original dataset and the dataset with the randomly-permuted metrics. The performance differences are measured in MCC, weighted Precision, and weighted Recall and are used as the effects of the group of metrics.

We follow the same steps to test the effects of all three groups of metrics on all the studied machine learning algorithms. After measuring the effects of all groups of metrics we apply the Scott-Knott ESD test to compare the effects of each group of metrics.

Table 5.3: The effects of groups of metrics on fixing effort prediction models. We highlight the highest effects in each algorithm.

Groups of metrics		**Median of performance drop percentage**					
		CNB	**KNN**	**MLP**	**RF**	**SVM**	**XGB**
Code metrics	MCC	17.8%	70.6%	64.2%	68.3%	33.1%	14.2%
	Precision	**14.2%**	10.4%	0%	0.13%	0%	0.05%
	Recall	7.3%	6.8%	**35.5%**	13.5%	**43.3%**	2.9%
Process metrics	MCC	9.3%	94%	31.7%	40.9%	42.7%	0.1%
	Precision	3.4%	**31.2%**	7.9%	21.9%	0.1%	2.2%
	Recall	**8.3%**	**41.5%**	4.3%	22.4%	23.0%	4.4%
Semantic metrics	MCC	**74.0%**	**95.7%**	**88.6%**	**100%**	**85.3%**	**35.5%**
	Precision	4.2%	19.4%	**32.1%**	**29.7%**	**20.4%**	**5.9%**
	Recall	7.3%	24%	6.4%	**49.0%**	11.0%	**8.8%**

Results: The semantic metrics impact the performance of our machine learning models the most. Table 5.3 shows the effects that each group of metrics has on the studied machine learning algorithms, measured using MCC, weighted

Table 5.4: The Scott-Knott ESD tests for comparing effects of groups of metrics on fixing effort prediction models

Performance metrics	Groups assigned by the Scott-Knott ESD tests	
	First group	Second group
MCC	Semantic metrics	Code metrics, process metrics
Precision	Semantic metrics	Code metrics, process metrics
Recall	Semantic metrics	Code metrics, process metrics

Precision, and weighted Recall. As shown in Table 5.3, the three groups of metrics all have positive effect on the performance of different machine learning algorithms, while semantic metrics have the most effect. Our results suggest that building eXtreme Gradient Boosting models without the semantic metrics can reduce the median MCC, weighted Precision, and weighted Recall by 35.5%, 5.9%, and 8.8%, respectively. The MCC, weighted Precision, and weighted Recall of the six machine learning algorithms drop an average of 27.4%, 16.7%, and 11.9% without using the semantic metrics.

Table 5.4 shows the Scott-Knott ESD test results of comparing the effects of the groups of metrics obtained from the six machine learning algorithms. Scott-Knott ESD test consistently ranks the semantic metrics in the most important group (i.e., first group) when measuring the effects using MCC, weighted Precision, and weighted Recall. The semantic metrics are derived from the directories, filenames, and comments from the methods to capture the functionalities of the buggy methods. The high importance of semantic metrics indicates that the functionality of a method can impact the time taken to fix bugs in the method.

Summary

The semantic metrics impact the performance of the fixing effort prediction models the most. Specifically, the MCC, weighted Precision, and weighted Recall of the six studied machine learning models drop an average of 27.4%, 16.7%, and 11.9% without using semantic metrics.

5.4 Threats to Validity

In this section, we discuss the threats to the validity of experiments conducted in this chapter.

Threats to external validity are related to the generalizability of our results. To ensure the generalizability of our proposed approach, we conduct experiments on 106 GitHub popular Java projects. These projects cover different domains. However, testing our approach on only GitHub open-source projects can introduce generalizability concerns for our conclusions. Studying software systems that are programmed in other languages and industry software systems can be useful to augment the generalizability of our approach.

In our experiments, we test our fixing effort prediction approach on the buggy methods that are predicted by CodeBERT. Evaluating our approach using the buggy methods that are predicted by other existing defect prediction approaches [54, 78, 108, 167, 168, 255, 256, 281] can be useful to check whether our approach can work well with different defect prediction approaches.

A limitation of our approach is to predict buggy methods in a new project or predict fixing effort categories of the predicted buggy methods with limited development history. This is because our approach relies on the defect fixing history of a project to

fine tune CodeBERT models to predict buggy methods and build machine learning models to predict fixing effort categories of the predicted buggy methods in the same project.

Threats to internal validity concern the uncontrolled factors that may affect our results. One internal threat to our results is that we give the same defect fixing effort category label for all the buggy methods in a same defect fixing commit. When developers work on a commit, there might be methods that can be modified quickly and methods that take a long time to modify. However, Git commits record only the time that developers commit changed files with no detailed information about the time developers spent on each method they change.

Another internal threat to our results is that we use keyword searching to identify the defect fixing commits following prior studies. However, there might be *false positive* defect fixing commits in our dataset. To check whether there are false positives (i.e., non defect fixing commits that are misclassified as defect fixing commits) in our dataset, we select a statistical sample of collected defect fixing commits and manually labeled it.

We use the resolution time of a defect to represent the fixing effort for the defect. However, developers do not necessarily start working on fixing a defect immediately when it gets assigned to them and the resolution time does not represent the exact amount of effort required to fix bugs. In our experiment, we predict the fixing effort categories of bugs instead of the exact fixing effort (e.g., amount of hours) to mitigate the impact of the noise in measuring the fixing effort of bugs.

5.5 Summary

In this chapter, we propose an approach that predicts the fixing effort categories of buggy methods that are predicted by a defect prediction approach. First, we predict buggy methods using the CodeBERT models. We apply three categories of metrics to measure the characteristics of buggy methods that are predicted by CodeBERT models and build machine learning models using the metrics as predictors to predict the fixing effort categories of the predicted buggy methods. We conduct extensive experiments on 106 popular Java projects to evaluate the performance of our fixing effort prediction approach.

Our results suggest:

- The eXtreme Gradient Boosting algorithm achieves the best performance in predicting the fixing effort categories of the predicted buggy methods with a median of 0.5 MCC, 0.81 Weighted Precision, and 0.8 Weighted Recall.
- The semantic metrics impact the performance of the fixing effort prediction models the most.

Chapter 6

Improving the Pull Requests Review Process Using Learning-to-rank Algorithms

The role of reviewers is key to maintain the effective review process of pull requests in pull-based development. However, the number of decisions that reviewers can make is far superseded by the increasing number of pull requests submissions. To help reviewers to perform more decisions on pull requests within their limited working time, in this chapter, we propose a learning-to-rank (LtR) approach to recommend pull requests that can be quickly reviewed by reviewers.

Chapter organization: In Section 6.1, we introduce this chapter. We describe our experiment setup and our results in Sections 6.2 and 6.3, respectively. In Section 6.4, we discuss the soundness of our chosen threshold, the experience levels of reviewers and the importance of the PRs recommended by our approach. The threats to validity are discussed in Section 6.5. Section 6.6 summarizes the chapter and suggests future work.

6.1 Introduction

The pull-based development model has become a standard for distributed software development and has been adopted in several collaborative software development platforms, such as GitHub, GitLab and Bitbucket [96]. Compared with other classic distributed development approaches (e.g., sending patches to development mailing lists [30]), the pull-based development model provides built-in mechanisms for tracking and integrating external contributions [199]. For example, with pull-based development model, some pull requests may be integrated with just one click, without manual intervention.

Under the pull-based development model, contributors can fetch their local copies of any public repository by *forking* and *cloning* them. Next, contributors can modify their clones to fix a bug or implement a new feature as they please. Ultimately, contributors may request to have their code changes merged into the central repository through pull requests (PRs) [251]. Once contributors submit PRs, they become available to reviewers (i.e., PRs undergo the opened state). If a PR is approved by reviewers and passes the integration tests, its respective code is merged into the central repository (i.e., the status of the PR changes to *merged*). Such collaborative software development platforms offer social media functionalities to allow contributors to easily interact with each other, such as following projects and communicating with reviewers.

As Gousios et al. [98] mentioned, the role of reviewers is key to maintain the quality of projects. Therefore, reviewers should carefully review PRs and decide whether a PR is worth integrating. Code reviewers should also communicate the modifications that are required before integrating a PR to external contributors. Nevertheless, reviewers

typically have to manage large amounts of open PRs simultaneously. Through our preliminary study of 74 projects hosted on GitHub, we observe that the workload (measured by the number of open PRs) of reviewers tends to rise while the decisions made by reviewers tend to remain roughly constant. As shown in Figure 6.1, the average number of open PRs is below five on January 2014. Overtime, the average number of open PRs keeps on increasing and reaches 30 on May 2016. However, the average number of reviewers and the average number of decisions made every day by reviewers remain roughly the same (e.g., an average of one decision per day). In fact, Steinmacher et al. [228] observed that 50% of PRs are submitted by casual-contributors, which considerably inflate the number of PR submissions that must be processed.

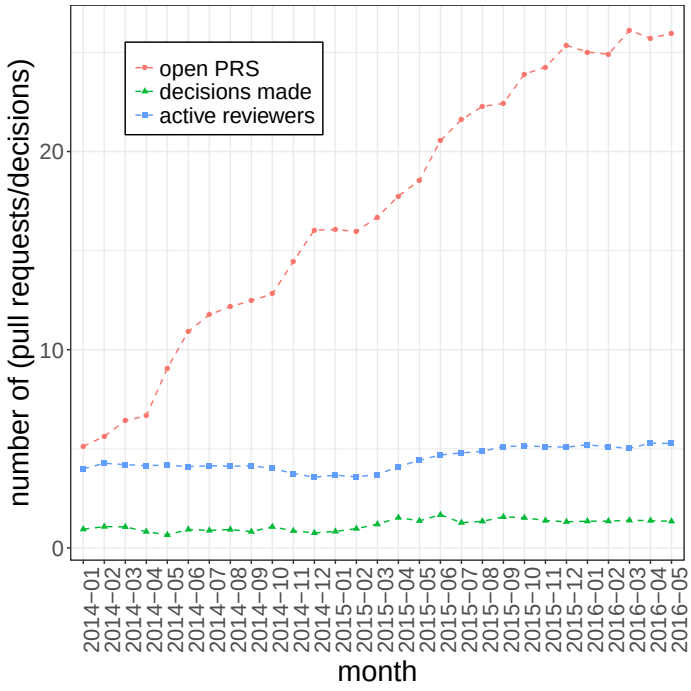


Figure 6.1: The average number of open PRs and the average number of decisions made every day among 74 Java projects on GitHub.

Gousios et al. [98] mentioned that prioritizing multiple PRs is important for reviewers when they face an increasing workload. The authors propose a PR prioritization tool, called PRioritizer, to automatically recommend the top PRs that reviewers should first deal with [250]. The tool can help reviewers select the PR that needs their immediate attention. To increase the decisions made by reviewers, the time taken by a reviewer to make decisions on PRs (either rejected or merged) should be considered when the tool recommends PRs to reviewers. In our preliminary study based on 74 projects, the median time taken by a reviewer to make decisions on PRs is 16.25 hours as shown in Figure 6.2. There are PRs that can be reviewed quickly (e.g., the minimum time taken to review a PR is 8 minutes), while some other PRs take a long time to review (e.g., the maximum time is more than 400 hours). More specifically, recommending reviewers PRs that can be quickly reviewed allows them to handle more contributions and give expedite feedback on PRs, which could be very useful when reviewers have only a limited time (e.g., half an hour or few minutes) to review PRs. The fast turn-around could ultimately improve the productivity of developers and reduce the waiting queue of open PRs so that contributors do not need to wait for a long time to receive feedback. Therefore, it is desirable to provide an approach that can recommend PRs that can be merged or rejected in a timely fashion. Instead of replacing the working practices, our approach aims to complement the existing practices of reviewers if they decide to take a number of quick decisions in a limited period of time. We adopt learning-to-rank (LtR) algorithms [112, 113] to rank PRs that are likely to be quickly merged or rejected. In particular, we address the following research questions.

RQ1. What learning-to-rank algorithm is the most suitable for ranking

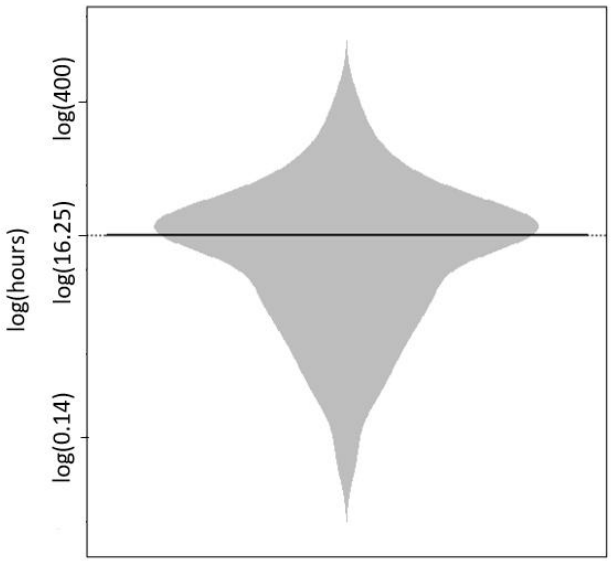


Figure 6.2: The time taken to make decisions on PRs among 74 Java projects on GitHub.

PRs?

We study six LtR algorithms including pairwise LtR algorithms and listwise LtR algorithms, such as RankBoost, MART, RankNet and the random forest. We find that the random forest based algorithm outperforms the other algorithms with respect to rank PRs that can receive quick decisions.

RQ2. Is our approach effective to rank PRs that can receive decisions quickly?

Gousios et al. [98] observed that there are two well-adapted prioritizing criteria among reviewers: first-in-and-first-out (FIFO) criterion and small-size-first criterion that recommends the PRs having the minimum source code churn. We compare the

performance of the random forest algorithm with the FIFO baseline and the small-size-first baseline. Our LtR approach outperforms both the FIFO baseline and the small-size-first baseline. Our results suggest that our LtR approach can help reviewers to make decisions regarding PRs more efficiently.

RQ3. What are the most significant metrics that affect the ranking?

To test the effect of each metric, we exclude a given metric and re-run the learning-to-rank approach without the metric. Then, we obtain the decreasement in the accuracy of LtR algorithm without that metric. The larger the decreasement, the more important a metric is. We find that metrics that represent the reputation of contributors (e.g., the previous PRs merged percentage) and the social connection of contributors (e.g., the number of followers and the previous interaction) affect the ranking the most.

6.2 Experiment Setup

In this section, we present the process for collecting data and extracting metrics for building LtR models. We also explain the analysis of correlated metrics. The overview of our approach is shown in Figure 6.3. We first select a set of experiment projects and collect the data of the selected projects (e.g., PRs, issues, commits). Next, we build LtR models on the 18 extracted metrics to recommend PRs that are likely to receive decisions quickly. We evaluate the performance of the LtR models and calculate the effect of each metric on ranking PRs.

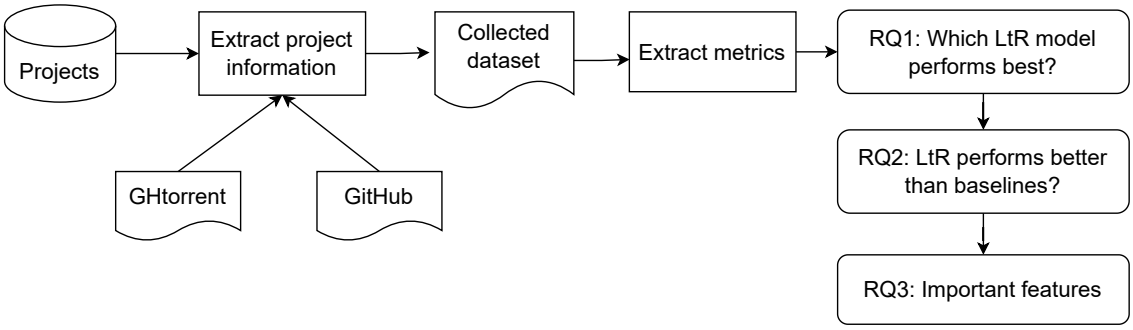


Figure 6.3: The overview of our approach.

6.2.1 Collecting data

As shown in Figure 6.3, we collect the data of our experiment from the GHTorrent database [92] and GitHub using the GitHub API.

Project Selection. To avoid working on personal, inactive or toy projects [143], we exclude projects containing less than 500 PRs. Our dataset contains data from November 2010 until May 2016. Given that we plan to analyze code quality metrics, we restrict our analyses to only one programming language (i.e., Java). This choice allows us to implement scripts that automatically compute, for example, the addition of code complexity or code comments, because we do not need to adapt to many different programming languages. Nevertheless, our approach can be adapted to other programming languages, provided that the metrics are properly extracted. We start with a set of 303 Java projects that contain more than 500 PRs. Then, we apply the following criteria to exclude projects from the initial selection:

- We exclude projects have been deleted from GitHub. For deleted projects, GHTorrent database still records its information (e.g., created time, and owner) but we cannot find the PRs of deleted projects, since all PRs have been cleaned.

- We exclude projects that are forked from other existing GitHub projects. As

Gousios et al. [93] mentioned, more than half of the GitHub repositories are forks of other repositories. To ensure that we study projects that receive external contributions, we only include original projects.

- The PRs of a project must be correctly labeled (*merged* or *closed*) for a recommender system to work correctly. However, as Steinmacher et al. [228] observed, many projects do not adopt pull based development. Instead, these projects prefer to merge contributions via git Command Line Interface (CLI). For example, through our manual analysis, we find PRs that are *closed* with comment *"Cherry picked. Thanks."* meaning that their commits were *merged*. As a result, the vast majority of PRs in such projects would be labeled as *closed* when their commits were actual *merged* into the main repository. We analyze the 94 remaining projects after excluding the deleted and forked projects. In the projects that do not adopt the pull based development, we find that the ratio of PRs labeled as *merged* over the total number of PRs is usually lower than 40%. In projects that use the pull based development, the ratio of merged PRs is around 73% [93]. To avoid studying noisy PRs, we exclude projects for which the merge ratio is less than 40% [93]. We discuss the impact of using different merge ratio thresholds in Section 6.4.2.

After filtering the projects, the number of our analyzed projects is reduced to 74.

Data Collection. After the project selection step, we collect the PRs, issue reports, comments and commits of the subject projects. We download the information by querying the GHTorrent (through Google BigQuery[1]) and searching GitHub

[1]http://ghtorrent.org/relational.html

repositories using the GitHub API. The GHTorrent database includes the meta information of PRs and commits, such as the id, the related project id and related PRs ids. However, information (e.g., textual information of PRs and the code difference information of commits) is not available in the GHTorrent database to compute our metrics (more details in Section 6.2.2). To collect such information, we use the GitHub API to search commits and PRs in GitHub repositories.

In short, our final dataset consists of 74 Java projects containing 100,120 PRs with 74,060 merged PRs and 26,060 rejected PRs.

6.2.2 Metrics Calculation

Given that we need to predict PRs that can receive a quick decision, we only collect the metrics that are available before the final decision (i.e., merged or rejected) of a PR. We select the metrics based on prior studies that investigate PRs [93, 98, 246, 250]. We categorize the selected metrics into four categories: source code metrics, textual metrics, contributor's experience metrics and contributor's social connection metrics.

Source code metrics. Our source code metrics describe the quality and the size of the modified source code in a PR. In total, we collect nine source code metrics, listed as follows.

- *test_inclusion (whether a contributor modifies test files in the PR).* Previous research [203] finds that PRs containing test files are more likely to be merged and reviewers perceive the presence of testing code as a positive indicator [98].

- *test_churn (the source code churn in test files).* Besides checking whether contributors change test files, we record the source code churns of the changes in test files. The source code churns measure the number of lines of code that

are modified. A high code churn may indicate a better quality PR because the contributor invests effort in the tests.

- *src_churn (the source code churns in other files that are not test files).* Prior research [262] reports that the size of a code patch plays an important role in both the acceptance of the patch and the acceptance time of the patch.

- *files_changed (the number of files touched in a PR).* We use the number of modified files as a measure of the scale of a PR. Large or small scale PRs might affect the time to merge or reject a PR.

- *num_commits (the number of commits included in a PR).* We use the number of commits as a measure of the scale of a PR. Large or small scale PRs in terms of the number of commits might affect the time to merge or reject a PR.

- *commits_files_changed (the number of total commits on files modified by a PR three months before submitting the PR).* Our goal is to check whether PRs modifying an active part of the system is more likely to be merged or rejected quickly.

- *bug_fix (whether a PR is an attempt to fix a bug).* Gousios et al. [98] finds that reviewers would consider reviewing the PRs that fix bugs before reviewing the PRs related to enhancements.

- *ccn_added, ccn_deleted (the cyclomatic complexity of newly added code and deleted code in a PR).* We use *ccn_added and ccn_deleted* to evaluate the source code quality of PRs because prior research [93, 98] finds that the source code quality of PRs could affect the decisions made on PRs.

- *comments_added (the number of comments that were introduced in the source code of a PR).* Well commented source code is easy to understand [240] and may accelerate the reviewing process.

Textual information metrics. Textual information metrics describe the general textual properties of a PR. We select the following three important metrics:

- *title_length (the number of words in the title of a PR).* A longer title may contain more useful information about a PR and may help reviewers easily understand a PR.

- *description_length (the number of words in the description of a PR).* A longer description may contain more meaningful information about a PR for reviewers to understand.

- *readability (the Coleman-Liau index [178] (CLI) of the title and description messages of a PR as well as its commit messages).* CLI has been adopted to measure the text readability in issue reports [118] and education material [50]. CLI represents the level of difficulty to comprehend a text, ranging from 1 (easy) to 12(hard). The equation for CLI is shown in Equation 6.1

$$CLI = 0.0588 * L - 0.296 * S - 15.8 \tag{6.1}$$

where L is the average number of characters per 100 words and S is the average number of sentences per 100 words.

Contributor's experience metrics. The contributor's experience metrics describe the experience of the contributors who submit PRs. We select two important metrics:

- *contributor_succ_rate*. The percentage of PRs from a contributor that have been merged before submitting the current PR.

- *is_reviewer*. Whether a contributor is also a reviewer in the same project. Code reviewers are much more experienced and more familiar with their projects and code reviewing process than the external contributors.

Contributor's social connection metrics. The contributor's social connection metrics represent the social connection that the contributor has on GitHub. We measure the social distance and the prior interactions of contributors.

- *social_distance* measures the social closeness between a contributor and a reviewer. If a contributor follows a reviewer, social_distance is set to 1, otherwise, it is 0.

- *prior_interaction* is used to count the number of events that a contributor has participated in the same project before submitting a PR [246]. Events include submissions of issue reports, PRs and comments.

- *followers* measures the number of followers that a contributor has. A high number of followers indicates a contributor's popularity and influence.

For each PR of the subject projects, we extract each aforementioned metric by performing a query in the collected datasets (e.g., counting the total number of PRs and the merged PRs of a contributor to calculate *contributor_cuss_rate*). Next, we store the metrics related to PRs into a table and train and test LtR models for each project.

6.2.3 Correlation and redundancy analysis

Highly correlated and redundant metrics can prevent us from measuring the effect of each metric. Therefore, we conduct correlation and redundancy analyses to remove highly correlated and redundant metrics.

We apply the Spearman rank correlation because it can handle non-normally distributed data [279]. We use the *cor()* function in R to calculate the correlation coefficient. If the correlation coefficient between two metrics is larger than or equal to 0.7, we consider the pair of metrics to be highly correlated and we select only one of the metrics. We intend to train and test LtR models on each project, which requires us to remove highly correlated metrics for each project. Instead obtaining the correlated pairs of metrics and select metrics for all projects manually, we run the Spearman rank correlation for each pair of metrics in all projects and record all possible highly correlated pairs. Then, we produce a *decision table* decides which metric should be kept in the occurrence of every possible correlation pair as shown in Table 6.1. For example, when *src_churn* and *ccn_addad* are highly correlated, we choose to keep *ccn_added* because we consider that the complexity of code is more important than the code churns. When building the LtR models for our projects, our approach reads the table and automatically chooses the metrics based on our produced *decision table*.

After performing the correlation analysis, we conduct a redundancy analysis on the metrics using the *redun()* function of the R *rms* package. Redundant metrics can be explained by other metrics in the data and do not aggregate values for models. We find that there exist no redundant metrics in our data.

Table 6.1: Decision table for highly correlated pairs

Metric1	Metric2	Choice
test_inclusion	*test_churn*	*test_churn*
src_churn	*files_changed*	*src_churn*
src_churn	*ccn_added*	*ccn_added*
src_churn	*ccn_deleted*	*ccn_deleted*
files_changed	*commit_file_changed*	*commit_file_changed*
ccn_added	*ccn_deleted*	*ccn_added*
src_churn	*comments_added*	*src_churn*
ccn_added	*comments_added*	*ccn_added*
test_churn	*src_churn*	*src_churn*
files_chnaged	*ccn_added*	*ccn_added*
contributor_succ_rate	*is_reviewer*	*contributor_succ_rate*
is_reviewer	*prior_interaction*	*prior_interaction*
src_churn	*commit_file_changed*	*commit_file_changed*
contributor_succ_rate	*prior_interaction*	*contributor_succ_rate*
test_churn	*files_changed*	*test_churn*
followers	*is_reviewer*	*is_reviewer*
test_churn	*ccn_added*	*test_churn*
test_churn	*comments_added*	*test_churn*
test_churn	*commit_file_changed*	*test_churn*
test_inclusion	*src_churn*	*test_inclusion*
test_inclusion	*files_changed*	*test_inclusion*
test_inclusion	*commit_file_changed*	*test_inclusion*
test_inclusion	*ccn_added*	*test_inclusion*
test_inclusion	*comments_added*	*test_inclusion*
files_changed	*comments_added*	*comments_added*
commit_file_changed	*ccn_added*	*commit_file_changed*
commit_file_changed	*comments_added*	*commit_file_changed*
followers	*prior_interaction*	*prior_interaction*
contributor_succ_rate	*followers*	*contributor_succ_rate*
followers	*ccn_deleted*	*ccn_deleted*
is_reviewer	*ccn_deleted*	is_reviewer

6.2.4 PRs labeling

Before training and testing the LtR models, the relevance between each PR and a query (e.g., a query refers to searching for the PRs that are most likely to be quickly merged, more details in Section 6.3.1) is described by (i) a label based on the decision made about the PR and (ii) the time taken to make the decision (as shown in Figure 6.4). Through our preliminary study, we find that the median time for reviewers to review PRs in each project ranges from 1.15 hours to 448.7 hours. We use the median time to review the PRs of a project as the threshold to split PRs into quickly reviewed PRs and slowly reviewed PRs. We use the median value because it is more robust to outliers [162]. Specifically, we use three relevance levels in querying PRs that are the most likely to be quickly merged:

- Relevance level 0 indicates PRs that are rejected.
- Relevance level 1 indicates PRs that take a long time to review and merge.
- Relevance level 2 indicates PRs that are quickly reviewed and merged.

Similarly,we use three relevance levels for identifying the PRs that are the most likely to be quickly rejected:

- Relevance level 0 indicates PRs that can be merged.
- Relevance level 1 designates PRs that are rejected but take a long time to be reviewed.
- Relevance level 2 specifies that a PR can be quickly rejected.

The number of PRs in each label is shown in Table 6.2.

Table 6.2: The number of PRs with different relevance levels. For querying the quickly merged PRs, (i) 0 indicates rejected PRs, (ii) 1 indicates slowly merged PRs and (iii) 2 indicates quickly merged PRs. For querying the quickly rejected PRs, (i) 0 indicates merged PRs, (ii) 1 indicates slowly rejected PRs and (iii) 2 indicates quickly rejected PRs.

	Querying the quickly merged PRs	**Querying the quickly rejected PRs**
Relevance level 0	26060	74060
Relevance level 1	35140	15322
Relevance level 2	38920	10738

6.3 Results

In this section, we describe the three research questions. We present the motivation, approach and results for each question.

6.3.1 RQ1. Which learning-to-rank algorithm is the most suitable for ranking PRs?

Motivation. To optimize the performance of our approach, it is crucial to select and deploy the most suitable LtR algorithm. We use the learning-to-rank framework, called RankLib[2], which consists of a set of LtR algorithms. Given the several options of LtR algorithms, it is important to evaluate the best LtR algorithm that suits our goal.

Approach. We use LtR models to rank PRs that allow a reviewer to make speedy decisions. LtR models have been widely applied in the information retrieval field [163, 288]. LtR models rank a set of documents (e.g., PRs) based on their relevance to a given query (e.g., PRs that can be quickly merged to the project). LtR

[2]https://sourceforge.net/p/lemur/wiki/RankLib/

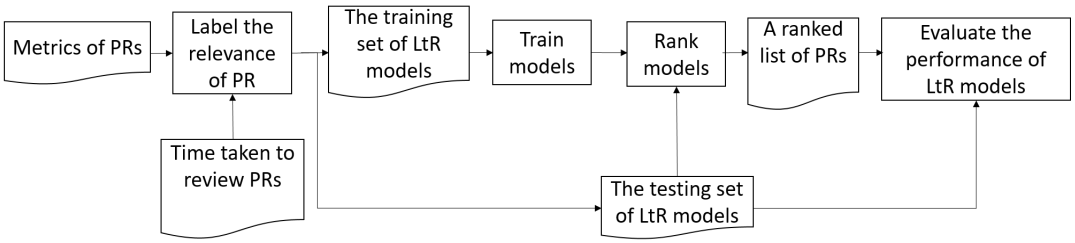


Figure 6.4: An overview of our ranking approach.

algorithms are supervised approaches and, as such, they have training and testing phases. The overall steps of our approach for ranking PRs are shown in Figure 6.4.

LtR algorithm selection. There are three categories of LtR algorithms based on their training process: *(1) pointwise algorithms* compute the absolute relevance score for each PR; *(2) pairwise algorithms* transform the ranking problem into a classification problem to decide which PR is more likely to receive a quick decision in a given pair of PRs; and *(3) listwise algorithms* take ranked lists of PRs as instances in training phase and learn the ranking model. In our approach, we explore 6 well-known and widely adopted pairwise (RankNet [38] and RankBoost [82]) and listwise (MART [83], Coordinate Ascent [183], ListNet [40] and random forest [36]) LtR algorithms. We exclude pointwise algorithms, since pairwise algorithms and listwise algorithms have been empirically proved to consistently outperform pointwise LtR algorithms [164].

Training phase. Every LtR model is trained using a set of queries $Q = \{q_1, q_2, \ldots, q_n\}$ and their related set of documents (i.e., studied PRs) $D = \{d_1, d_2, \ldots, d_n\}$. More specifically, a query refers to searching for the PRs that are most likely to be quickly merged. Each document d is represented by one studied PR, $P = \{p_1, p_2, \ldots, p_n\}$. For each query q, the related PRs are labeled with their relevance to query q. During the training process, for each query, the LtR algorithm

computes the relevance between each PR and the query using the metric vector V_s of the PR. In this chapter, we use one query (i.e., q_1) to identify PRs that are most likely to be quickly merged and another query (i.e., q_2) to identify PRs that are most likely to be quickly rejected. We define d, r, q_1, q_2 and V_s as follows:

- Document d is a PR.
- Relevance r is the likelihood of d being quickly merged or rejected (depending on the query).
- Query q_1 is a query to identify the PRs that are the most likely to be quickly merged.
- Query q_2 queries which PRs are the most likely to be quickly rejected.
- Metric vector V_s denotes a set of metrics used to build the LtR models as discussed in Section 6.2.2.

Testing phase. We measure the performance of the LtR model for ranking PRs that are most likely to be quickly merged and rejected separately, since ranking quickly merged and rejected PRs are trained using different sets of queries.

Given that software projects evolve over time [189], we need to consider the time-sensitive nature of our data. For example, we cannot test models using PRs that were closed before the PRs that we use in our training data (i.e., predicting the past). For this reason, we use a time-sensitive validation approach to evaluate the LtR models as shown in Figure 6.5. First, we sort our PRs based on their closing time. Then, we split our data into 5 folds $F = \{f_1, f_2, \ldots, f_5\}$. Each f_i is split into a training set t_i and a testing set τ_i. In the first iteration, we train a ranking model using the t_1

training set of PRs, while we use the τ_1 testing set as a query to test the model. In the second iteration, a new ranking model is trained using the t_1, τ_1, t_2 sets (i.e., all PRs before τ_2) and tested using the τ_2 set as a query. This process continues until a ranking model is tested upon the τ_5 set. Finally, we compute the average performance of the ranking models in these five iterations.

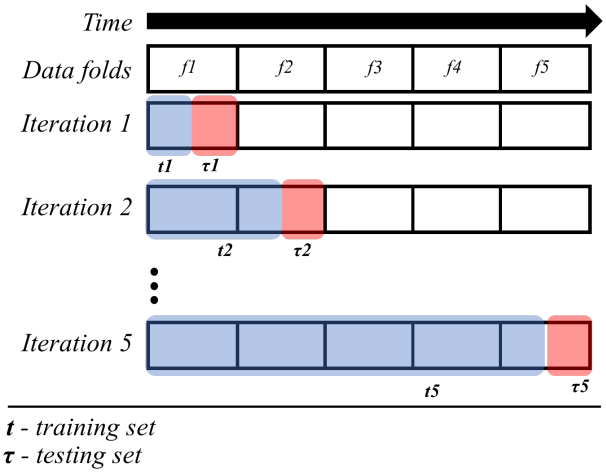


Figure 6.5: An overview of our time-sensitive evaluation

To evaluate the performance of an LtR model, we use the normalized discounted cumulative gain at position k ($NDCG@k$), which is a well-adapted measure of ranking quality. $NDCG$ is a weighted sum of degree of relevance of the ranked PRs [258]. Search engine systems also use a cut-off top-k version of $NDCG$, referred to as $NDCG@k$ [258]. We opt for using $NDCG@k$ because the precision at position k ($P@k$) is not suitable for multiple labels [194]. In addition, Recall does not suit our case because hundreds of open PRs are likely to be merged quickly, while in our approach we rank only 20 PRs for reviewers to work on.

We first calculate the discounted cumulative gain at position k ($DCG@k$) [36]. DCG has an explicit position discount factor in its definition (i.e., the $\frac{1}{\log_2(1+j)}$ as

shown in equation 6.2). PRs with a high relevance r but having a low ranking would negatively affect the DCG metric. $DCG@k$ is calculated as follows [36]:

$$DCG@k = \sum_{j=1}^{k} \frac{2^{r_j} - 1}{\log_2 (1 + j)} \tag{6.2}$$

where r_j is the relevance label of a PR in the j_{th} position in the ranking list.

For the example shown in Table 6.3, $DCG@1 = 3$, $DCG@2 = 3$, $DCG@3 = 3.5$. Then, the $DCG@k$ is normalized using the optimal $DCG@k$ value ($IDCG@k$) to get the $NDCG@k$ as shown in Equation 6.3.

$$NDCG@k = \frac{DCG@k}{IDCG@k} \tag{6.3}$$

Table 6.3: An example of the ranking result of three PRs of the MovingBlocks/Terasology project for query q_1 : *quickly merged PRs*

Rank	**Pull request**	**Relevance**
1	Develop - Block Manifestor cleanup + some...	quickly merged
2	PullRequest - Cleanup	rejected
3	AddedEclipse-specific content to .gitignore...	slowly merged

Table 6.4: The optimal rank of PRs in Table 6.3

Rank	**Pull request**	**Relevance**
1	Develop - Block Manifestor cleanup + some...	quickly merged
2	AddedEclipse-specific content to .gitignore...	slowly merged
3	PullRequest - Cleanup	rejected

For the example shown in Table 6.3, the optimal ranking is shown in Table 6.4, and $IDCG@3 = \frac{2^2-1}{\log_2 (1+1)} + \frac{2^1-1}{\log_2 (1+2)} + \frac{2^0-1}{\log_2 (1+3)} = 3.63$ and $NDCG@3 = \frac{DCG@3}{IDCG@3} = \frac{3.5}{3.63} = 0.96$.

We apply each LtR algorithm to rank the PRs of each project following the time-sensitive validation method as shown in Figure 6.5. Then, we compute the $NDCG@k$ metric for $k = 1, \ldots, 20$ for each project. After applying one LtR algorithm to all projects, we obtain a distribution of the $NDCG$ metric at each ranking position k (1,. . . ,20). For each ranking position k, we draw a beanplot[3] to show the performance of each algorithm at position k (e.g., the beanplot of $NDCG@1$ of RankNet algorithm as shown in Figure 6.6).

To compare the performance of different LtR models in the same ranking k position, we use the Cliff's delta to measure the magnitude of differences between performance distributions of LtR models at the same position. The larger the delta value, the larger the difference between two distributions. We use the *cliff.delta()* method of the *effsize* package in R to calculate the Cliff's delta.

Results. **The random forest model outperforms the other LtR models in ranking both quickly merged PRs and quickly rejected PRs.** As shown in Figure 6.6, for ranking quickly merged PRs, the median $NDCG@k$ values of the random forest model is almost 0.8, while the median values of the other models are all around 0.6. In addition, the magnitude of the differences in performance between the random forest model and the other models are small and medium in all the k positions as shown in Table 6.5.

Regarding ranking quickly rejected PRs, the performances of the six models are not promising as shown in Figure 6.7. The most probable reason is that the ranking models are not well trained, since the quickly rejected PRs only account for 10% of the total set of PRs as shown in Table 6.2. Another possible reason is that there may still exist some noise in the rejected PRs in our dataset although we have excluded

[3]https://cran.r-project.org/web/packages/beanplot/vignettes/beanplot.pdf

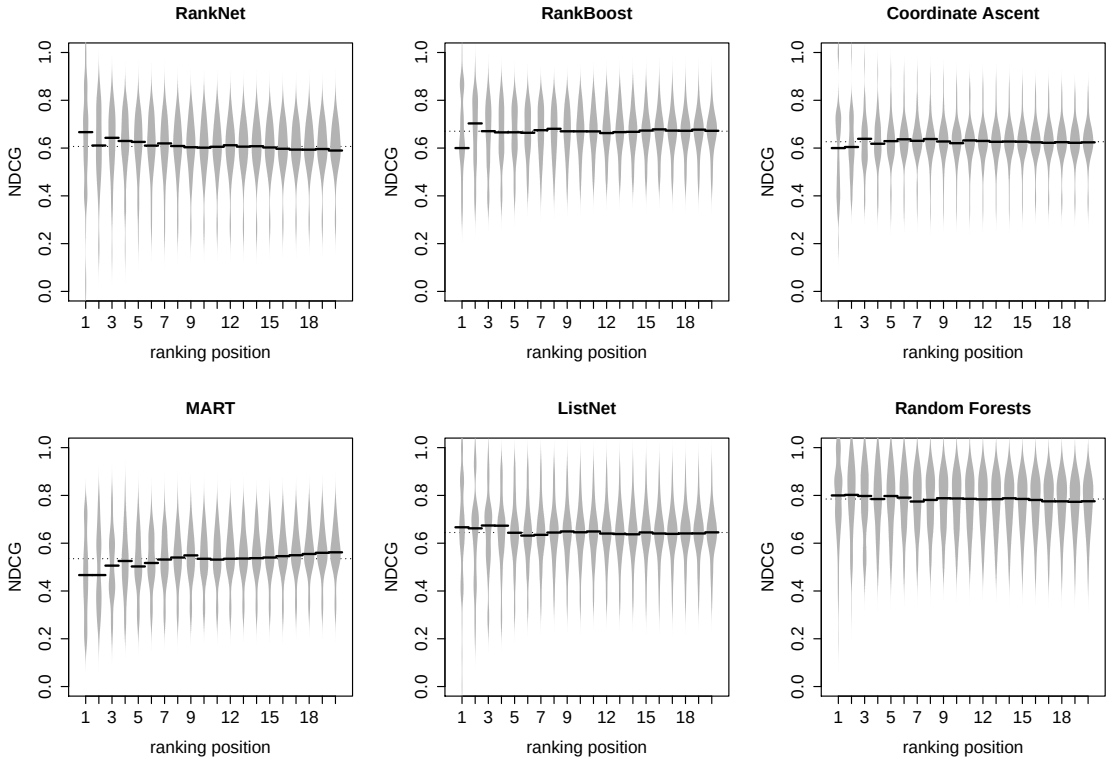


Figure 6.6: The performance of the LtR models to rank PRs that can be quickly merged

Table 6.5: The median Cliff's Delta estimate of all k positions and the Cliff's Delta magnitude

Algorithm	**Cliff's Delta Estimate**	**Cliff's Delta Magnitude**
RankNet	0.334	medium
RankBoost	0.198	small
Coordinate Ascent	0.341	medium
MART	0.451	medium
ListNext	0.350	medium

projects with a merge ratio lower than 0.4. Nevertheless, the random forest model performs better than the other 5 models when ranking quickly rejected PRs.

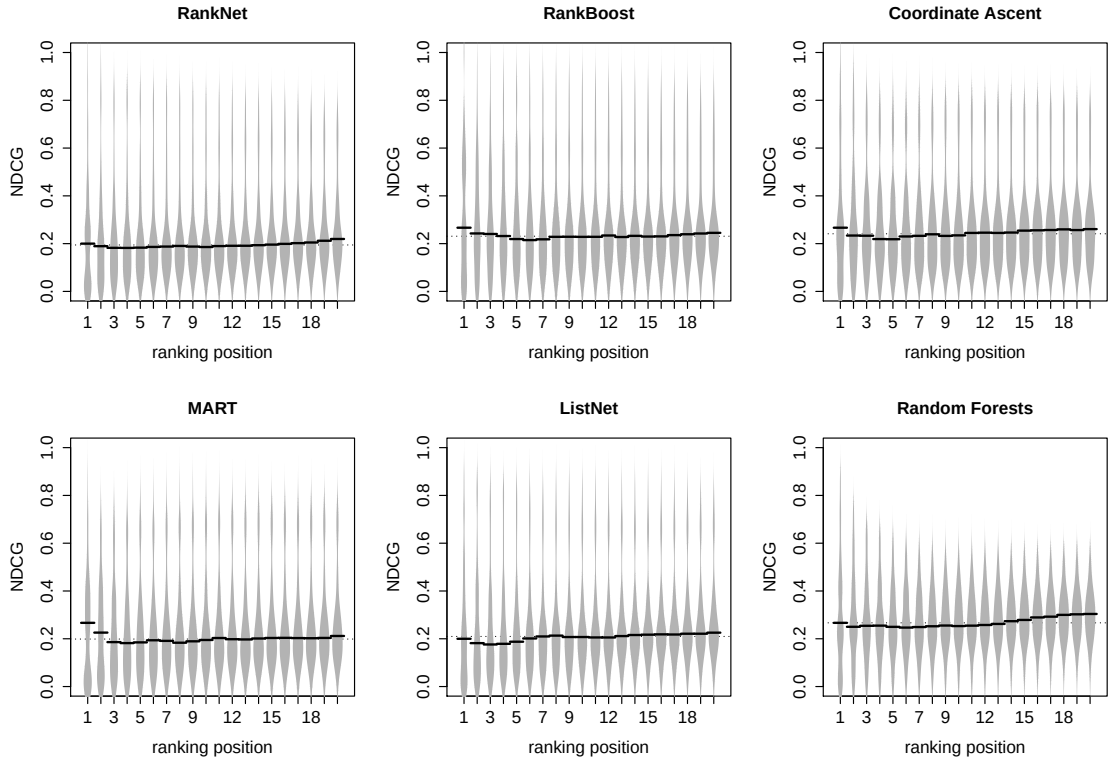


Figure 6.7: The performance of the LtR models to rank PRs that can be quickly rejected

Summary

The random forest model performs the best when ranking PRs based on their likelihood of being quickly merged.

6.3.2 RQ2. Is our approach effective to rank PRs that can receive decisions quickly?

Motivation. After selecting the best performing LtR model in our approach, we need to verify the effectiveness of our proposed LtR model by comparing it with the existing prioritizing criteria that are studied by Gousios et al. [98].

Approach. There is no universal conclusion on how reviewers prioritize PRs in practice. Nevertheless, Gousios et al. [98] observed that there are two well-adapted prioritizing criteria among reviewers through their large-scale qualitative study [98]. The two criteria are listed as follows:

- *First-in-first-out.* As mentioned in their research, many reviewers prefer a first-in-first-out prioritization approach to select the PRs [98], based on the age of the PRs. In this criterion, reviewers would first focus on the PRs that come earlier than others.

- *Small-size-first.* Besides the age of the PR, reviewers use the size of the patch (source code churn of the PRs) to quickly review small and easy-to-review contributions and process them first.

We build two baselines based on the aforementioned two criteria, i.e., the first-in-first-out (FIFO) baseline and the small-size-first baseline.

We test the two baselines using the same time-sensitive approach that we use to test the performance of the random forest model (see the approach section of RQ1). Similarly, we measure the performance of each baseline to rank PRs that are the most likely to be quickly merged and rejected separately. Next, we use Equation 6.3 to calculate the $NDCG@k$ metric. Similar to Section 6.3.1, after running the two baselines in each project, we use beanplots and Cliff's delta measures to show the performance of two baselines and compare them with the random forest model.

Results. **The random forest model outperforms both the FIFO and small-size-first baselines for ranking the PRs that can be either quickly merged or quickly rejected.** Figures 6.8 and 6.9 show the performance of the random forest model, the FIFO baseline and small-size-first baseline for ranking the

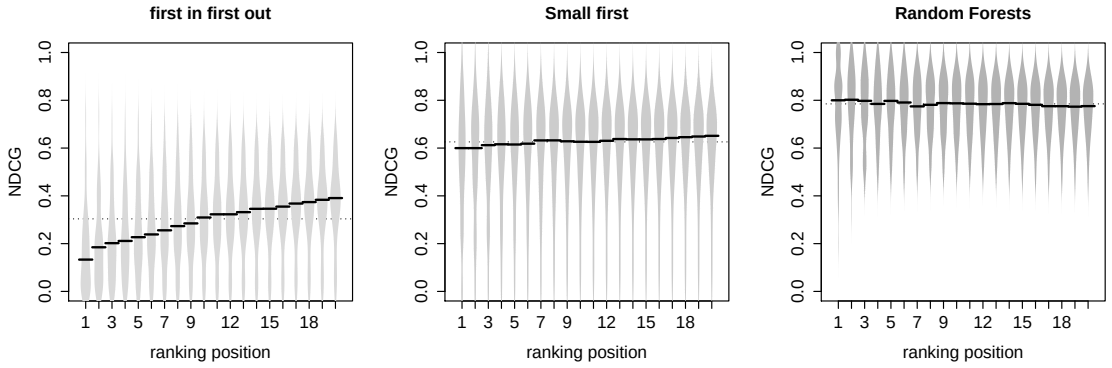


Figure 6.8: The performance of our LtR approach and two baselines to rank PRs that can be quickly merged.

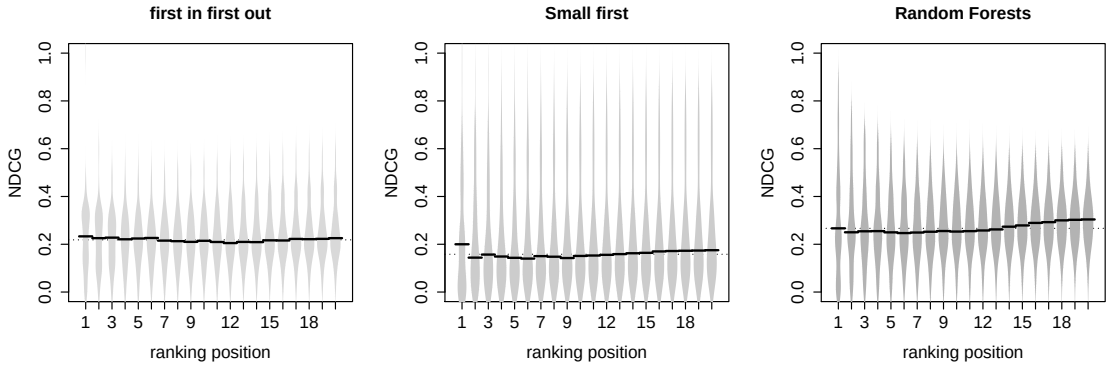


Figure 6.9: The performance of our LtR approach and two baselines to rank PRs that can be quickly rejected.

PRs that can receive quick decisions. We observe a large Cliff's delta (median Cliff's Delta estimate = 0.60) between the random forest model and the FIFO baseline. Thought the Cliff's delta between the random forest model and the small-size-first baseline is small (median Cliff's Delta estimate = 0.18), the random forest model can help reviewers to merge more contributions in a shorter time as shown in the following ranking example.

To show the application of our ranking approach, we run the random forest LtR

model and the small-size-first baseline on the same set of PRs of the libgdx/libgdx project. The libgdx/libgdx project has been attracting external contributions since 2012. There are more than 70 open PRs waiting for reviewers to review every day in this project. The ranking list of the random forest LtR model and the ranking list of the small-size-first baseline are shown in Tables 6.6 and 6.7, respectively. Based on the top 10 PRs in both tables, we can observe that our LtR approach ranks PRs that can be merged in a shorter time at the top positions when compared with the small-size-first approach.

Table 6.6: Example ranking top 10 PRs results of the LtR random forest model

Position	**Pull request**	**label**	**Observed review time (hours)**
1	Move DebugDrawer	2	0.013
2	Fix linker flags...	2	0.628
3	error msg for...	2	10.416
4	Javadoc minor typos	2	0.518
5	Update CHANGES	2	1.02
6	Everyone misspells...	2	0.035
7	iOS: Force linked...	1	50.233
8	Implement material...	2	0.001
9	Move gwtVersion to...	2	0.041
10	Update fetch.xml	2	0.037

We also observe that the small-size-first baseline performs as good as the other LtR models (e.g., RankNet, RankBoost and ListNet) when ranking the PRs that can be quickly merged or rejected. This observation suggests that the small-size-first baseline is, to a certain extent, optimized for ranking PRs that can receive quick decisions. Breaking large PRs into several smaller PRs (whenever possible) could increase the likelihood of quickly deciding on a PR.

Table 6.7: Example ranking top 10 PRs results of the small-size-first baseline

Position	Pull request	label	Observed review time (hours)
1	Removed Unused variable...	2	4.988
2	Merge pull request #2...	0	0.011
3	update	0	0.007
4	Merge pull request #1...	0	0.028
5	Very Minor Update...	2	8.424
6	Gradle: Enforced OpenGL...	2	12.169
7	Update CHANGES	2	0.054
8	MathUtils: Fixed isEqual...	2	0.157
9	Stop IDE's thinking...	0	2.483
10	Everyone misspells...	2	2.483

Summary

The LtR using the random forest model outperforms the FIFO and small-size-first baselines when ranking PRs that can be quickly merged or rejected. By considering the high performance of the LtR model, reviewers could use the model to aid them in merging more contributions in a shorter time.

6.3.3 RQ3. What are the most significant metrics that affect the ranking?

Motivation. In our experiment, we leverage many metrics that capture different aspects of a PR. In this research question, we investigate the effect of each metric on ranking the PRs that can be quickly merged and observe the most influential metrics in the LtR model. This is important to better understand how a PR can receive a decision more quickly with a small set of metrics that have the most significant impact on the ranking.

Approach. To test the effect of each metric, we first exclude a given metric from the training dataset. Next, we build the random forest model on the new dataset and

test its performance. Finally, we measure the difference in the performance in terms of $NDCG@k$ regarding ranking PRs that can be quickly reviewed of the ranking model without the metric. The larger the drop in the performance, the higher the effect of the tested metric in the LtR model.

Table 6.8: Effects of each ranking metric on PRs.

Test metric	**k=1**	**k=3**	**k=5**
social_distance	4.13%	2.67%	2.17%
followers	3.20%	2.34%	1.99%
num_commits	1.96%	0.00%	0.04%
description_length	1.82%	2.08%	2.00%
ccn_deleted	1.36%	0.35%	0.39%
readability	1.19%	0.65%	1.41%
commit_file_changed	0.09%	2.05%	1.57%
bug_fix	0.47%	0.00%	0.72%
comments_added	0.46%	-0.08%	1.37%
contributor_succ_rate	-0.04%	0.05%	1.00%
status	-0.27%	-0.82%	0.17%
src_churn	-0.62%	0.18%	-0.25%
ccn_added	-1.17%	-0.86%	-0.24%
test_churn	-1.22%	-0.38%	-0.18%
files_changed	-1.85%	0.04%	-0.02%
prior_interaction	-3.26%	-1.45%	-0.91%

Results. **The metrics related to the social connection of a contributor are the most significant metrics.** Table 6.8 shows the effects of each metric. The percentages in Table 6.8 show the decreases in the performance measure $NDCG@k$ of the random forest model after removing the tested metric.

Without using *soical_distance* and *followers*, the performance of the random forest model decreases the most (4.13% and 3.20% off in the first position respectively). Both *soical_distance* and *followers* are related to the social connections of a contributor. The first two most important metrics suggest that the social connections of

the contributor play an influential role in the decision making process of PRs. Our approach could be used to attract the attention of reviewers when a PR is submitted by a contributor with a good social connection. Another two important metrics are *num_commits* and *ccn_deleted*, which represent the scale of the PRs and the source code quality of PRs respectively, as we explained in Section 6.2.2. We also find that the experience of the contributors is not as significant as the social connection of contributors and source code quality of PRs, especially for *prior_interaction*, based on the negative values shown in Table 6.8. When we remove *prior_interaction* from the LtR model, the performance of the LtR model increases by 3.26% at the first position.

In addition, we observe that *description_length* and *readability* are important, which suggests that the title and description texts of PRs have an important effect in the reviewing process. The negative effect of *src_churn* on LtR model indicates that the churn in PRs is not an important prioritization criterion in all of the cases for which a quick decision could be taken.

GitHub contributors can leverage the important metrics found in our LtR model to attract the reviewers' attention. For example, a new external contributor should seek interactions with reviewers in the same project to build a social connection with the reviewers. The external contributor is encouraged to follow reviewers on GitHub and comment on PRs. By doing so, the external contributor may engage in discussions with reviewers before submitting a PR to the project. In addition, contributors should write meaningful descriptions, titles and commits messages of PRs, so that reviewers feel inclined to work on the submissions of the contributors.

Summary

The social connection of a contributor is more important than the source code quality of a PR and the experience of the contributor. And reviewers should not only consider the source code churn of the PRs as a single criterion for prioritizing reviews.

6.4 Discussion

In this section, we discuss the the implication of our finding, soundness of our threshold to exclude noise PRs. Next, we discuss whether the PRs that our approach recommends are in fact useful and not trivial (i.e., simple modifications to documentation). We also discuss whether our approach is suitable to reviewers of diverse levels of experience.

6.4.1 Implication of our finding

Through our experiment, we find that the social connections of developers are the most important metrics. Thus, the pull requests from popular and influential developers are more likely to be reviewed quickly. However, the popularity of developers should not always be the primary factor in prioritizing pull requests. To better support the development of software, the criticality and the contribution of the pull requests should be considered. For example, pull requests that fix critical security vulnerabilities should have high priority and be reviewed by developers quickly after its submission.

6.4.2 Threshold to remove noise PRs

We conduct a sensitivity analysis to verify that the merge ratio threshold of 0.4 enables us to eliminate most of wrongly labeled PRs and keep correctly labeled PRs. We apply three different merge ratio thresholds (i.e., 0.4, 0.5, and 0.6). We do not include threshold of 0.7 because the normal merge ratio of PRs is around 0.7 and using 0.7 would exclude projects where the PRs are correctly labeled [93]. For each PR merge ratio threshold, we conduct the same experiment and evaluation approach covered in Sections 6.2 and 6.3. Figures 6.10 and 6.11 show the performance of our model of using different thresholds to exclude projects where PRs are incorrectly labeled.

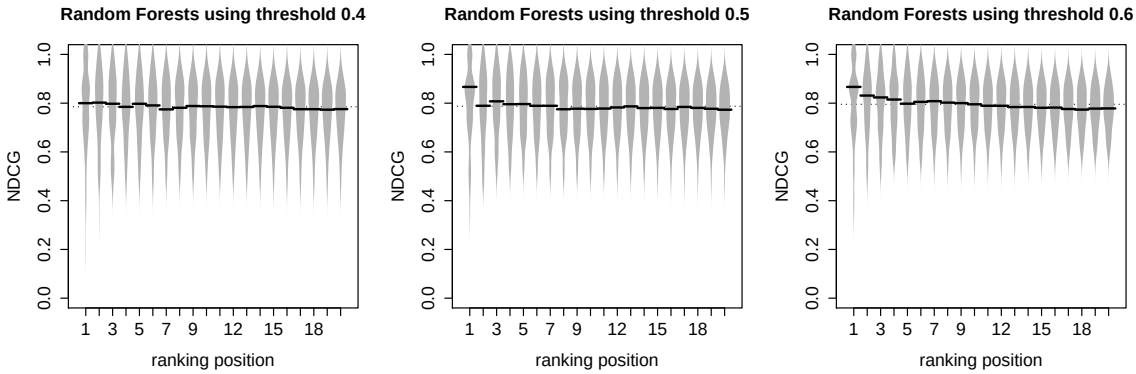


Figure 6.10: The performance of our LtR approach to rank PRs that can be quickly merged under three different thresholds.

In Figures 6.10 and 6.11, we observe that the performance of our approach keeps at the same level when we increase the merge ratio threshold. The Cliff's difference delta is negligible between the performances. The performance of recommending quickly merged PRs increases slightly (median Cliff's Delta estimate = 0.05) by raising the PRs merge ratio threshold, while the performance of recommending quickly closed

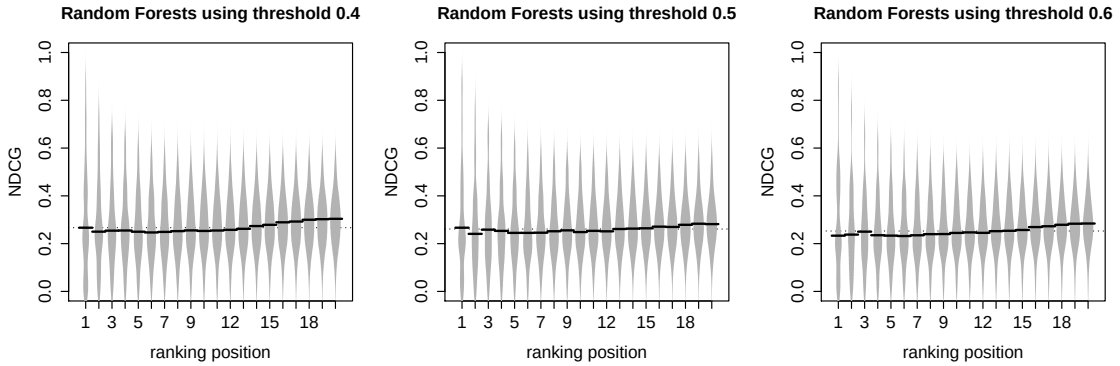


Figure 6.11: The performance of our LtR approach to rank PRs that can be quickly rejected under three different thresholds.

PRs decreases slightly (median Cliff's Delta estimate = 0.04). A possible reason for the slight change in the performance is that a higher merge ratio threshold can exclude more wrongly labeled PRs, but also remove correctly labeled *closed* PRs. Therefore, we infer that the merge ratio threshold of 0.4 is able to eliminate most of wrongly labeled PRs while keeping correctly labeled PRs.

6.4.3 Types of the recommended PRs

The paramount goal of code reviewing is to maintain or improve the quality of the projects [98]. Therefore, it is critical to examine the impact of using our approach on improving the quality of projects. If our approach only recommends PRs that modify only project documentation, the benefit of improving the quality of a project is trivial compared with recommending bug fixing or feature implementation PRs. Although reviewers can always follow their own priority of PRs, our approach would not be useful to reviewers if we only recommended trivial PRs (e.g., documentation PRs).

Our approach recommends a total number of 1,480 PRs among the 74 projects in the experiment project set. We take a statistical sample of 305 PRs with 95%

Table 6.9: Types of pull requests [116]

Types of Pull Requests	**Description**
Bug fix	A pull request fixes one or more bugs.
Build	A pull request focuses on changing the build or configuration system flies (e.g., pom.xml files).
Clean up	A pull request cleans up the unused attributes, methods, classes.
Documentation	A pull request is designed to update documentation of a system (e.g., method comments).
Feature implementation	A pull request adds or implements a new feature.
Enhancement	A pull request performs activities common during a maintenance cycle (different from bug fixes, yet, not quite as radical as adding new features).
Test	A Pull request related to the files that are required for testing.
Others	A Pull request related to language translation, platform specific, token rename and source code refactoring.

confidence level and with a 5% confidence interval. We randomly select 305 PRs from the entire population of recommended PRs. In prior research [116], different categories for commits are proposed to group commits. In our approach, we focus on PRs which consist of commits. Thus, we adopt the commits categories mentioned in research [116] to classify our PRs. We classify a PR based on the information contained in the description and title. We manually classify each PR in the sample into one of the types shown in Table 6.9. The number and percentage of PRs in each type are shown in Table 6.10.

Table 6.10: The number of PRs in each type

Pull request categories	**Number of pull request**	**Percentage of pull request**
Enhancement	78	26%
Bug fix	65	22%
Feature implementation	42	14%
Build	31	10%
Test	28	9%
Documentation	22	7%
Clean up	12	4%
Others	24	8%

Table 6.10 indicates that the majority of recommended PRs are related to enhancement, bug fixing, and new feature implementation. Conversely, documentation PRs only account for 7% of PRs recommended by our approach. A possible reason for the low percentage of documentation PRs is that our LtR model is built on the previous reviewed PRs. Reviewers usually prefer to review enhancement, bug fixing, and new feature implementation PRs rather than trivial PRs (i.e., PRs that do not aggregate much value for the project). Therefore, we conclude that PRs recommended by our approach could help reviewers bring more positive contributions to the project and improve the quality of projects.

6.4.4 PRs reviewed by reviewers of different experience levels

In this section, we check whether reviewers having different experience levels are interested in various types of PRs. This investigation is important because it checks whether our approach is suitable for reviewers of diverse levels of experience. For example, new reviewers may focus only on simple reviews (i.e., documentation or

clean up PRs), while experienced reviewers are more willing to review complex PRs.

For each PR within the sample that we use in Section 6.4.3, we measure the experience of the reviewers who reviewed the PR. We select the following three metrics to measure the experience of reviewers:

- *prs_reviewed*. The number of PRs that one reviewer has reviewed before reviewing a specific PR.
- *prs_submitted*. The number of PRs that one reviewer has submitted to the project before reviewing a specific PR.
- *commits*. The number of commits that one reviewer has preformed before reviewing a specific PR.

We extract these three experience metrics by querying our collected dataset as explained in Section 6.2.1. Next, we group the experience levels of reviewers based on the type of PRs. After obtaining a set of experience levels for the reviewers of each PR type, we use the standard deviation to measure the variation on the experience levels (shown in Table 6.11).

We observe that, the standard deviation is high (the smallest value is 119.73) for each type of PR, which indicates that all PRs of each type are reviewed by reviewers with different levels of experience. Besides standard deviation, we also perform a Kruskal-Wallis H test [152] to test whether the experience levels of reviewers for different types of PRs have similar values. The null and alternative hypotheses are:

- *null hypothesis* H_0: The experience levels of reviewers of different types of PRs are similar.

Table 6.11: The standard deviation of experiences of reviewers

Types of pull request	**Standard deviation of experience of reviewers**		
	Measured in *prs_reviewed*	**Measured in *prs_submitted***	**Measured in *commits***
Enhancement	295.25	225.32	818.59
Bug fix	295.19	177.31	641.26
Feature implementation	278.98	143.49	709.07
Build	325.42	237.34	725.38
Test	317.27	124.12	474.32
Documentation	274.98	215.52	894.90
Clean up	140.00	119.73	515.61
Others	253.77	207.45	851.70

- *alternative hypothesis* H_1: The experience levels of reviewers of different types of PRs are significantly different.

Table 6.12: The result of Kruskal-Wallis H test

Reviewers experiences	**p-value**
Measured in *prs_reviewed*	0.31
Measured in *prs_submitted*	0.90
Measured in *commits*	0.80

Based on the result of the Kruskal-Wallis H test shown in Table 6.12, the p-values of the three experience metrics are 0.31, 0.80 and 0.90, which are far above 0.05. Thus, we cannot reject the null hypothesis that all types of PRs are reviewed by reviewers of all levels of experience.

Therefore, based on the experiment results of standard deviation and Kruskal-Wallis H test, we observe that there is no apparent relationship between the experience levels of reviewers and the types of PRs that they are willing to review.

6.5 Threats to validity

In this section, we discuss the threats to the validity of experiments conducted in this chapter.

Threats to external validity concern whether the results of our approach are able to be generalized for other situations. In our experiment, we include PRs from 74 GitHub Java projects. We filter out projects to ensure that our analyzed projects are well-developed and popular among GitHub contributors. However, projects in other programming languages (e.g., Python and Javascript) may have different reviewing process for PRs, and our findings might not hold true for non-Java projects [228]. For example, the metrics that capture the characteristics of source code might be important for reviewing PRs in other language projects. To address this threat, further research including other language projects is necessary to obtain more generalized results.

Threats to internal validity concern the uncontrolled factors that may affect the experiment results. One of the internal threats to our results is that we assume that the behavior of reviewers does not change over time. As Gousios et al. [98] and Steinmacher et al. [228] mentioned, checking whether PRs follow the current developing goal of a project is the top priority for reviewers when evaluating PRs. Also, it is possible that, at different stages of a project, the project enforces different developing policies (e.g., reviewers may focus more on bug fixing contributions than feature enhancement when the release time is approaching). However, it is challenging to capture all of these metrics (i.e., the changing goals of the project) without closely contacting the core contributors of each project. In spite of this challenge, we attempt to cover metrics from four dimensions: source code metrics, social connection metrics,

experience metrics, and textual information metrics. Besides the changing developing policies, different reviewers may have different expertise. Ideally, our approach should be customized to recommend PRs based on the preferences of each reviewer. However, the majority of PRs in a project are reviewed by a few numbers of reviewers [273], which means we cannot obtain enough reviewing history for most reviewers. It is problematic to train a LtR model on each reviewer. Therefore, we only use objective metrics related to PRs to build a generalized approach that can work for all reviewers. Our goal is not to replace reviewers' prioritization criteria of PRs, but rather to be another tool at their hands to improve their working environment. Additionally, there may exist noise (i.e., incorrectly labeled PRs) in the rejected PRs, which can have a negative impact on the performance of the LtR models as mentioned in Section 6.3.1.

Threats to construct validity concern whether the setup and measurement in this chapter reflect real-world situations. In this chapter, we treat the time interval between the submitted time of a PR and the close time of the PR as the reviewing time. However, the time interval is a rough estimation of the actual time spent by reviewers on evaluating the PRs. In practice, there is always a delay for reviewers to notice the PRs after its submission and the delay should be counted in the reviewing time. Unfortunately, there is no information indicating the exact time when a reviewer started reviewing a PR. On the other hand, the delay time might reflect the priority of PRs. The longer a PR waits, the lower the priority of that PR might be. Therefore, we find it reasonable to use the overall time interval as an approximation of reviewing time, which is used to separate the studied PRs into quickly reviewed PRs and slowly reviewed PRs. In addition, it is difficult to quantify the exact amount of time saved from reviewers due to the unknown delay for reviewers to notice the PRs.

Nevertheless, we observed that our approach can be useful for reviewers to review more PRs within a limited time.

6.6 Summary

In this chapter, we propose a LtR model to recommend quick-to-review PRs to help reviewers to make more speedy decisions on PRs during their limited working time. We use 18 metrics to build LtR models and we use six different LtR algorithms, such as ListNet, RankNet, MART and random forest. We conduct empirical studies on 74 Java projects to compare the performances of the six LtR algorithms. We compare the best performing algorithm against two baselines obtained from previous research regarding pull requests prioritization: the first-in-and-first-out baseline and the small-size-first baseline.

The results show:

- The random forest LtR algorithm outperforms other five well adapted LtR algorithms to rank quickly merged pull requests.
- The random forest LtR algorithm performs better than both the FIFO and the small-size-first baselines, which means our LtR approach can help reviewers make more decisions and improve their productivity.
- The contributor's social connections and contributor's experience are the most influential metrics to rank pull requests that can be quickly merged.

Chapter 7

Conclusions and Future Work

High performance is a critical factor to achieve and maintain the success of a software system. Performance anomalies represent the performance degradation issues (e.g., slowing down in system response times) of software systems at run-time. Performance anomalies can cause a dramatically negative impact on users' satisfaction. In this thesis, we propose a framework to help developer and operator teams to avoid performance anomalies at the run-time, and capture and fix performance defects at the development phase.

In the following sections, we summarize the contributions of this thesis and suggest the potential research opportunities that may advance our work.

7.1 Contributions

The main contributions of the thesis are listed below.

(1) Proposing an approach to predict performance anomalies in software systems at run-time (Chapter 3). Researchers have proposed various approaches to automatically monitor software systems and detect anomalies at run-time [133, 137, 187, 188, 218, 253]. However, the delay in detecting anomalies and

taking corrective actions can result in the violations of Service Level Objective (SLO) or system failures. To prevent performance anomalies from happening, we propose an approach that can predict performance anomalies in software systems and raise anomaly warnings in advance. Specifically, our proposed approach first uses Long-Short Term Memory (LSTM) neural networks to capture the normal behaviors of software systems. Then, our approach predicts performance anomalies at run-time by checking the early deviations from the normal behaviors that are expected from the LSTM neural networks. Using our approach, operators may receive warnings of performance anomalies in advance and take proactive actions to prevent the performance anomalies from happening.

Summary

Our results show that 1) our approach outperforms the anomaly prediction baselines and predicts various performance anomalies with 97-100% precision and 80-100% recall; 2) our approach can achieve lead times varying from 20 to 1,403 seconds (i.e., 23.4 minutes), which are sufficient for automatic prevention approaches to take proper actions and prevent the predicted anomalies from happening; and 3) our approach achieves 95-100% precision and 87-100% recall with lead times that vary from 2 to 846 seconds (i.e., 14.1 minutes) for predicting the anomalies that are caused by the five real-world performance defects.

(2) Providing an approach to predict a large variety of performance defects (Chapter 4). Performance defects are non-functional defects that can cause performance anomalies at run-time. Prior studies [53, 86, 191, 195, 270, 280] have proposed different approaches to find performance defects via identifying anti-patterns in

source code. However, each type of performance defect has its own unique anti-pattern and requires a different approach to analyze the source code. Adopting a different approach to detect each type of performance defect is time-consuming. Moreover, prior approaches [53, 86, 191, 195, 280] cannot predict performance defects that do not follow the identified anti-patterns. To help developers identify performance defects at an early stage of the development phase, we provide an approach that can predict various types of performance defects. We propose performance code metrics to measure characteristics of poor performance code instead of measuring restricted performance anti-patterns [53, 86, 191, 195, 280]. Then, we build performance defect predictors using machine learning models such as Random Forest, eXtreme Gradient Boosting, and Linear Regressions. Developers may use our approach to identify performance defects at an early stage of software development phase and prevent the bugs from causing performance anomalies.

Summary

Our case study results show that 1) our approach predicts various types of performance defects at a file level with a median of 0.84 AUC, 0.21 Precision-Recall AUC (PR-AUC), and 0.38 Matthews correlation coefficient (MCC); 2) the proposed performance code metrics are the most important metrics in the studied machine learning models for predicting performance defects; and 3) our approach can predict additional performance defects that are not covered by the anti-patterns proposed in the prior studies.

(3) Providing an approach to predict the fixing effort for predicted defects (Chapter 5). Predicting fixing effort for defects (both performance defects

and non-performance defects) can help project teams prioritize defects and coordinate effort during bug triaging. Existing studies [8, 28, 88, 118, 160, 176, 261, 285] propose approaches to analyze the descriptions and attributes of new issue reports and estimate the effort for fixing the defects described in the issue reports. However, existing studies require information from issue reports to predict the fixing effort of defects. Therefore, the existing studies cannot provide estimation about the fixing effort for the predicted defects which have not been reported by developers. To address such limitation, we propose an approach that predicts the fixing effort categories of buggy methods that are predicted to have any types of defects (e.g., performance defects and non-performance defects) by a defect prediction approach. First, we use CodeBERT models to predict buggy methods. We apply three categories of metrics to measure the characteristics of predicted buggy methods and build machine learning models using the metrics as predictors to predict the fixing effort categories of the predicted buggy methods.

Summary

Our case study results show that 1) our approach predicts the fixing effort categories of predicted buggy methods with a median of 0.5 MCC, 0.8 of weighted precision, and 0.8 weighted recall; and 2) the semantic metrics impact the performance of the fixing effort prediction models the most.

(4) Understanding the workload of reviewers and proposing an approach to prioritize pull requests (Chapter 6). In pull-based development, developers submit pull requests to fix defects (e.g., performance defects and non-performance defects). Reviewers play an important role in maintaining the high

quality of code changes in pull requests. However, little is known about the workload (e.g, the number of pull requests waiting to be reviewed) of reviewers. We observe that the workload (number of pull requests waiting to be reviewed) for reviewers tends to increase while the number of decisions made on pull requests remains roughly the same over time. Thus, it is important to help reviewers to make more speedy decisions on pull requests during their limited working time. We propose an approach to recommend quick-to-review pull requests to reviewers. We use four categories of metrics to measure pull requests and use the metrics as predictors to build learning-to-rank models to recommend pull requests.

Summary

Our case study results show that 1) our approach outperforms the pull requests prioritization baselines and achieves a median of 0.8 normalized discounted cumulative gain (NDCG) values for recommending pull requests that can be quickly reviewed to reviewers; and 2) the majority of recommended pull requests are related to performance enhancement, bug fixing, and new feature implementation.

7.2 Future Work

Our proposed approaches make a positive impact on automatically predicting performance anomalies at both run-time and development phase of software systems and helping developers fix the defects for predicted anomalies. However, the thesis can be extended from multiple perspectives. In this section, we outline the promising extensions of the thesis for the future work.

- **Predicting performance anomalies and performance defects for software systems written in other programming languages**. Our proposed approaches that predict performance anomalies and performance defects are specific to software systems implemented in Java programming languages. Our proposed performance anomaly prediction scheme can analyze the collected resource usages of a running program regardless of the implementation programming languages. However, we use a toolkit, called IBM Javametrics that can monitor resource usages of only running Java programs. Similarly, our proposed performance defect prediction scheme can be adapted to other programming languages assuming the code metrics are properly calculated. In our experiments, we implemented scripts to calculate metrics for only Java programs. However, there are other popular programming languages, e.g., Python and Javascript. Thus, the future research is encouraged to extend our approaches to predict performance anomalies and defects for software systems written in other programming languages.

- **Automatically learning code characteristics of performance defects**. To predict various types of performance defects, we propose a set of performance code metrics to measure the code characteristics of performance defects. Through the experiments, we observe that the proposed performance code metrics improve the predictive power of our models the most. However, our performance code metrics cannot cover all types of performance defects, thus, it is likely that the proposed performance code metrics fail to capture the code characteristics of some performance defects (e.g., performance defects related

to system-specific inefficient function logic). Thus, the future work could investigate the feasibility of using the unsupervised learning algorithms to automatically learn and select the important code features from historical performance defect fixing code changes.

- **Conducting cross-project performance defects prediction**. Our performance defect prediction approach predicts performance defects in a project by leveraging various historical data in the project. For new projects that do not have sufficient historical data to train our prediction approach, our approach may not work. To address such limitation, the future work could provide cross-project performance defect prediction approaches, i.e., training prediction machine learning models using historical data from projects with sufficient historical data and predicting performance defects in new projects.

- **Providing reasons behind the performance defect prediction**. Our proposed performance defect prediction approach identifies the files that contain performance defects, but our proposed approach does not provide information (e.g., reasons why our approach predicts the files as buggy or exact lines of code where the defects are in the files) to guide developers to fix the performance defects. To help developers fix the identified performance defects, the future work could identify performance defects at a fine grained level of the source code (e.g., method level) and provide reasons for the prediction results.

- **Predicting the possible performance impact of predicted performance defects**. Our proposed approaches could predict a large set of performance defects at the development phase of software systems. Due to time-to-market, it

is challenging for developers to localize and address all the predicted performance defects before software releases. performance defects can have different impact on the software systems. For example, a memory leak bug can eventually take all the memory spaces and break down software systems, while performance regression bugs (i.e., inefficient or unoptimized source code) can only make software systems perform slowly or use more calculation resources. To help developers prioritize defects with potentially high performance impact, the future work could propose approaches to estimate the performance impact of predicted defects and recommend predicted defects with high performance impact to developers.

Bibliography

[1] SciTools (2020). Understand 6.0. URL: https://www.scitools.com/ [cited January 15, 2021].

[2] Walid AbdelMoez, Mohamed Kholief, and Fayrouz M Elsalmy. Improving bug fix-time prediction model by filtering out outliers. In *2013 The International Conference on Technological Advances in Electrical, Electronics and Computer Engineering (TAEECE)*, pages 359–364. IEEE, 2013.

[3] Alluxio. Commit 58c24eb8 in project alluxio. URL: https://github.com/Alluxio/alluxio/commit/58c24eb80baca6c2f4efbb9fb7349fd80c1d3c0a [cited December 02, 2021].

[4] Alluxio. Journaledgroup.java in project alluxio. URL: https://github.com/Alluxio/alluxio/blob/master/core/server/common/src/main/java/alluxio/master/journal/JournaledGroup.java [cited December 02, 2021].

[5] Douglas G Altman, Berthold Lausen, Willi Sauerbrei, and Martin Schumacher. Dangers of using optimal cutpoints in the evaluation of prognostic factors. *JNCI: Journal of the National Cancer Institute*, 86(11):829–835, 1994. URL: http://doi.org/10.1093/jnci/86.11.829, doi:10.1093/jnci/86.11.829.

[6] Erik Altman, Matthew Arnold, Stephen Fink, and Nick Mitchell. Performance analysis of idle programs. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 739–753, 2010. URL: `http://dx.doi.org/10.1145/1869459.1869519`, `doi:10.1145/1869459.1869519`.

[7] Amazon. Amazon ec2 instance types. URL: `https://aws.amazon.com/ec2/instance-types/` [cited September 5, 2020].

[8] Prasanth Anbalagan and Mladen Vouk. On predicting the time taken to correct bug reports in open source projects. In *2009 IEEE International Conference on Software Maintenance*, pages 523–526. IEEE, 2009.

[9] Android. Commit 465088ed in project android_platform_frameworks. URL: `https://android.googlesource.com/platform/frameworks/base/+/465088ed2f4591d08738b2306f213c5149b3484b` [cited December 02, 2021].

[10] Android. Connectivityservice.java in project android_platform_frameworks. URL: `https://android.googlesource.com/platform/frameworks/base/+/465088ed2f4591d08738b2306f213c5149b3484b/services/core/java/com/android/server/ConnectivityService.java` [cited December 02, 2021].

[11] Antlr. Java grammars in antlr. URL: `https://github.com/antlr/grammars-v4/tree/master/java` [cited December 02, 2021].

[12] Aosp-mirror. Commit 5abc71b2 in project platform_frameworks_base. URL: `https://github.com/aosp-mirror/platform_frameworks_base/commit/5abc71b27b5cd08148a277a8f378bbb5f4029835` [cited May 07, 2022].

[13] Aosp-mirror. Commit aea1bdec in project platform_packages_apps_settings. URL: https://github.com/aosp-mirror/platform_packages_apps_settings/commit/aea1bdec2d20753bbb64b53ac95b347395877493 [cited May 07, 2022].

[14] Apache. Apache hadoop randomtextwriter application. URL: https://hadoop.apache.org/docs/r1.2.1/api/org/apache/hadoop/examples/RandomTextWriter.html [cited November 27, 2019].

[15] Apache. Apache hadoop system. URL: http://hadoop.apache.org/ [cited November 27, 2019].

[16] Apache. Benchmarkthroughput.java in hadoop. URL: https://github.com/apache/hadoop/blob/3427bc1380ab4455a311c1848a83a966996bbc95/hadoop-hdfs-project/hadoop-hdfs/src/test/java/org/apache/hadoop/hdfs/BenchmarkThroughput.java [cited December 02, 2021].

[17] Apache. Issue 13514 in hdfs. URL: https://issues.apache.org/jira/browse/HDFS-13514 [cited December 02, 2021].

[18] Apache. Jobimpl.java in project hadoop. URL: https://github.com/Jerry-Xin/hadoop/blob/master/hadoop-mapreduce-project/hadoop-mapreduce-client/hadoop-mapreduce-client-app/src/main/java/org/apache/hadoop/mapreduce/v2/app/job/impl/JobImpl.java [cited December 02, 2021].

[19] Erik Arisholm, Lionel C Briand, and Eivind B Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction

models. *Journal of Systems and Software*, 83(1):2–17, 2010. URL: `http://dx.doi.org/10.1016/j.jss.2009.06.055`, `doi:10.1016/j.jss.2009.06.055`.

[20] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004.

[21] Shaeela Ayesha, Muhammad Kashif Hanif, and Ramzan Talib. Overview and comparative study of dimensionality reduction techniques for high dimensional data. *Information Fusion*, 59:44–58, 2020.

[22] Alberto Bacchelli, Marco DAmbros, and Michele Lanza. Are popular classes more defect prone? In *International Conference on Fundamental Approaches to Software Engineering*, pages 59–73. Springer, 2010. URL: `http://dx.doi.org/10.1007/978-3-642-12029-9_5`, `doi:10.1007/978-3-642-12029-9_5`.

[23] Adrian Bachmann, Christian Bird, Foyzur Rahman, Premkumar Devanbu, and Abraham Bernstein. The missing links: bugs and bug-fix commits. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 97–106, 2010.

[24] Vipin Balachandran. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 931–940. IEEE, 2013.

[25] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *OSDI*, volume 4, pages 18–18, 2004.

[26] Stefan Berner, Roland Weber, and Rudolf K Keller. Observations and lessons learned from automated testing. In *Proceedings of the 27th international conference on Software engineering*, pages 571–579, 2005.

[27] Kanishka Bhaduri, Kamalika Das, and Bryan L Matthews. Detecting abnormal machine characteristics in cloud infrastructures. In *2011 IEEE 11th International Conference on Data Mining Workshops*, pages 137–144. IEEE, 2011.

[28] Pamela Bhattacharya and Iulian Neamtiu. Bug-fix time prediction models: can we do better? In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 207–210, 2011.

[29] Suparna Bhattacharya, Mangala Gowri Nanda, Kanchi Gopinath, and Manish Gupta. Reuse, recycle to de-bloat software. In *European Conference on Object-Oriented Programming*, pages 408–432. Springer, 2011. URL: `http://dx.doi.org/10.1007/978-3-642-22655-7_19`, `doi:10.1007/978-3-642-22655-7_19`.

[30] Christian Bird, Alex Gourley, and Prem Devanbu. Detecting patch submission and acceptance in oss projects. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 26. IEEE Computer Society, 2007.

[31] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code! examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 4–14, 2011. URL: `http://dx.doi.org/10.1145/2025113.2025119`, `doi:10.1145/2025113.2025119`.

[32] Bisq-network. Commit 55b070f9 in project bisq. URL: `https://github.com/bisq-network/bisq/commit/55b070f9556977aa6ec4ebf878498a03b44f2f0` [cited May 07, 2022].

[33] BluSunrize. Issue 2468 in immersiveengineering. URL: `https://github.com/BluSunrize/ImmersiveEngineering/issues/2468` [cited December 02, 2021].

[34] Peter Bodík, Moises Goldszmidt, and Armando Fox. Hilighter: Automatically building robust signatures of performance behavior for small-and large-scale systems. In *SysML*, 2008.

[35] Kendrick Boyd, Vitor Santos Costa, Jesse Davis, and C David Page. Unachievable region in precision-recall space and its effect on empirical evaluation. In *Proceedings of the... International Conference on Machine Learning. International Conference on Machine Learning*, volume 2012, page 349. NIH Public Access, 2012. URL: `http://digital.library.wisc.edu/1793/61736`.

[36] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[37] Bugzilla. Bugzilla home page. URL: `https://www.bugzilla.org/` [cited December 02, 2021].

[38] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. Learning to rank using gradient descent. In *Proceedings of the 22nd international conference on Machine learning*, pages 89–96. ACM, 2005.

[39] Rodrigo N Calheiros, Kotagiri Ramamohanarao, Rajkumar Buyya, Christopher Leckie, and Steve Versteeg. On the effectiveness of isolation-based anomaly detection in cloud data centers. *Concurrency and Computation: Practice and Experience*, 29(18):e4169, 2017.

[40] Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. Learning to rank: from pairwise approach to listwise approach. In *Proceedings of the 24th international conference on Machine learning*, pages 129–136. ACM, 2007.

[41] Sucheta Chauhan and Lovekesh Vig. Anomaly detection in ecg time signals via deep long short-term memory networks. In *2015 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pages 1–7. IEEE, 2015.

[42] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002. doi:10.1613/jair.953.

[43] Jinfu Chen and Weiyi Shang. An exploratory study of performance regression introducing code changes. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 341–352. IEEE, 2017. URL: http://dx.doi.org/10.1109/ICSME.2017.13, doi:10.1109/ICSME.2017.13.

[44] Jinfu Chen, Weiyi Shang, and Emad Shihab. Perfjit: Test-level just-in-time prediction for performance regression introducing commits. *IEEE Transactions on Software Engineering*, 2020. doi:10.1109/TSE.2020.3023955.

[45] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E Hassan, Mohamed Nasser, and Parminder Flora. Detecting performance anti-patterns for applications developed using object-relational mapping. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1001–1012, 2014. URL: http://dx.doi.org/10.1145/2568225.2568259, doi:10.1145/2568225.2568259.

[46] Yiqun Chen, Stefan Winter, and Neeraj Suri. Inferring performance bug patterns from developer commits. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 70–81. IEEE, 2019. URL: http://dx.doi.org/10.1109/ISSRE.2019.00017, doi:10.1109/ISSRE.2019.00017.

[47] Ludmila Cherkasova, Kivanc Ozonat, Ningfang Mi, Julie Symons, and Evgenia Smirni. Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 452–461. IEEE, 2008.

[48] Davide Chicco and Giuseppe Jurman. The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation. *BMC genomics*, 21(1):1–13, 2020. doi:10.1186/s12864-019-6413-7.

[49] Ira Cohen, Steve Zhang, Moises Goldszmidt, Julie Symons, Terence Kelly, and Armando Fox. Capturing, indexing, clustering, and retrieving system history. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 105–118. ACM, 2005.

[50] Marc Colaco, Peter F Svider, Nitin Agarwal, Jean Anderson Eloy, and Imani M Jackson. Readability assessment of online urology patient education materials. *The Journal of urology*, 189(3):1048–1052, 2013.

[51] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. Input-sensitive profiling. *ACM SIGPLAN Notices*, 47(6):89–98, 2012. URL: `http://dx.doi.org/10.1145/2345156.2254076`, `doi:10.1145/2345156.2254076`.

[52] Ting Dai, Daniel Dean, Peipei Wang, Xiaohui Gu, and Shan Lu. Hytrace: a hybrid approach to performance bug diagnosis in production cloud infrastructures. *IEEE Transactions on Parallel and Distributed Systems*, 30(1):107–118, 2018. URL: `http://dx.doi.org/10.1109/TPDS.2018.2858800`, `doi:10.1109/TPDS.2018.2858800`.

[53] Ting Dai, Jingzhu He, Xiaohui Gu, Shan Lu, and Peipei Wang. Dscope: Detecting real-world data corruption hang bugs in cloud server systems. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 313–325, 2018. URL: `http://dx.doi.org/10.1145/3267809.3267844`, `doi:10.1145/3267809.3267844`.

[54] Hoa Khanh Dam, Trang Pham, Shien Wee Ng, Truyen Tran, John Grundy, Aditya Ghose, Taeksu Kim, and Chul-Joo Kim. A deep tree-based model for software defect prediction. *arXiv preprint arXiv:1802.00921*, 2018.

[55] Augusto Born De Oliveira, Sebastian Fischmeister, Amer Diwan, Matthias Hauswirth, and Peter F Sweeney. Perphecy: performance regression test selection made simple but effective. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 103–113. IEEE, 2017. doi:10.1109/ICST.2017.17.

[56] Daniel J Dean, Hiep Nguyen, Xiaohui Gu, Hui Zhang, Junghwan Rhee, Nipun Arora, and Geoff Jiang. Perfscope: Practical online server performance bug inference in production cloud computing infrastructures. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–13, 2014. URL: http://dx.doi.org/10.1145/2670979.2670987, doi:10.1145/2670979.2670987.

[57] Daniel J Dean, Hiep Nguyen, Peipei Wang, Xiaohui Gu, Anca Sailer, and Andrzej Kochut. Perfcompass: Online performance anomaly fault localization and inference in infrastructure-as-a-service clouds. *IEEE Transactions on Parallel and Distributed Systems*, 27(6):1742–1755, 2015. URL: http://dx.doi.org/10.1109/TPDS.2015.2444392, doi:10.1109/TPDS.2015.2444392.

[58] Daniel J Dean, Peipei Wang, Xiaohui Gu, William Enck, and Guoliang Jin. Automatic server hang bug diagnosis: Feasible reality or pipe dream? In *2015 IEEE International Conference on Autonomic Computing*, pages 127–132. IEEE, 2015. URL: http://dx.doi.org/10.1109/ICAC.2015.52, doi:10.1109/ICAC.2015.52.

[59] Daniel Joseph Dean, Hiep Nguyen, and Xiaohui Gu. Ubl: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems.

In *Proceedings of the 9th international conference on Autonomic computing*, pages 191–200. ACM, 2012.

[60] Sampath Deegalla and Henrik Bostrom. Reducing high-dimensional data by principal component analysis vs. random projection for nearest neighbor classification. In *2006 5th International Conference on Machine Learning and Applications (ICMLA'06)*, pages 245–250. IEEE, 2006.

[61] Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. Untangling fine-grained code changes. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 341–350. IEEE, 2015.

[62] Zishuo Ding, Jinfu Chen, and Weiyi Shang. Towards the use of the readily available tests from the release pipeline as performance tests. are we there yet? In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1435–1446. IEEE, 2020. URL: `https://ieeexplore-ieee-org.proxy.queensu.ca/abstract/document/9284085`.

[63] Bruno Dufour, Barbara G Ryder, and Gary Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive java applications. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 59–70, 2008.

[64] Marco DAmbros, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4-5):531–577, 2012. URL: `http://dx.doi.org/10.1007/s10664-011-9173-9`, `doi:10.1007/s10664-011-9173-9`.

[65] Bradley Efron. Estimating the error rate of a prediction rule: improvement on cross-validation. *Journal of the American statistical association*, 78(382):316–331, 1983. URL: http://dx.doi.org/10.1080/01621459.1983.10477973, doi:10.1080/01621459.1983.10477973.

[66] Bradley Efron and Robert J Tibshirani. *An introduction to the bootstrap*. CRC press, 1994. URL: https://www.routledge.com/An-Introduction-to-the-Bootstrap/Efron-Tibshirani/p/book/9780412042317.

[67] Elastic. Bulkshardresponse.java in project elasticsearch. URL: https://github.com/elastic/elasticsearch/blob/master/server/src/main/java/org/elasticsearch/action/bulk/BulkShardResponse.java [cited December 02, 2021].

[68] Elastic. Commit 6b51d85c in project elasticsearch. URL: https://github.com/elastic/elasticsearch/commit/6b51d85cbde8e0ddea020dcccb1e798dcb4ef27a [cited May 07, 2022].

[69] Elastic. Commit c5315744 in project elasticsearch. URL: https://github.com/elastic/elasticsearch/commit/c531574407c5547fc742b168f61c03fd00b0c530 [cited December 02, 2021].

[70] Elastic. Docwriteresponse class in project elasticsearch. URL: https://www.javadoc.io/doc/org.elasticsearch/elasticsearch/6.0.1/org/elasticsearch/action/DocWriteResponse.html [cited December 02, 2021].

[71] Elastic. Publicationtransporthandler.java in project elasticsearch. URL: `https://github.com/elastic/elasticsearch/blob/master/server/src/main/java/org/elasticsearch/cluster/coordination/PublicationTransportHandler.java` [cited December 02, 2021].

[72] Elastic. Rally. URL: `https://github.com/elastic/rally` [cited December 9, 2019].

[73] ElasticSearch. Elasticsearch. URL: `https://www.elastic.co` [cited November 27, 2019].

[74] ElasticSearch. Elasticsearch reference. URL: `https://www.elastic.co/guide/en/elasticsearch/reference/5.3/modules-threadpool.html` [cited November 27, 2019].

[75] ElasticSearch. Elasticsearch reference. URL: `https://www.elastic.co/` [cited January 21, 2021].

[76] Úlfar Erlingsson, Marcus Peinado, Simon Peter, Mihai Budiu, and Gloria Mainar-Ruiz. Fay: extensible distributed tracing from kernels to clusters. *ACM Transactions on Computer Systems (TOCS)*, 30(4):1–35, 2012.

[77] Facebook. Commit 2258ba13 in project buck. URL: `https://github.com/facebook/buck/pull/2553/commits/2258ba13d8c4ed7bce41975571609a05c3939bba` [cited May 07, 2022].

[78] Guisheng Fan, Xuyang Diao, Huiqun Yu, Kang Yang, and Liqiong Chen. Deep semantic feature learning with embedded static metrics for software defect prediction. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pages 244–251. IEEE, 2019.

[79] Mostafa Farshchi, Ingo Weber, Raffaele Della Corte, Antonio Pecchia, Marcello Cinque, Jean-Guy Schneider, and John Grundy. Contextual anomaly detection for a critical industrial system based on logs and metrics. In *2018 14th European Dependable Computing Conference (EDCC)*, pages 140–143. IEEE, 2018.

[80] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[81] King Chun Foo, Zhen Ming Jiang, Bram Adams, Ahmed E Hassan, Ying Zou, and Parminder Flora. Mining performance regression testing repositories for automated performance analysis. In *2010 10th International Conference on Quality Software*, pages 32–41. IEEE, 2010.

[82] Yoav Freund, Raj Iyer, Robert E Schapire, and Yoram Singer. An efficient boosting algorithm for combining preferences. *Journal of machine learning research*, 4(Nov):933–969, 2003.

[83] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[84] Song Fu. Performance metric selection for autonomic anomaly detection on cloud computing systems. In *2011 IEEE Global Telecommunications Conference-GLOBECOM 2011*, pages 1–5. IEEE, 2011.

[85] Song Fu, Jianguo Liu, and Husanbir Pannu. A hybrid anomaly detection framework in cloud computing using one-class and two-class support vector machines. In *International Conference on Advanced Data Mining and Applications*, pages 726–738. Springer, 2012.

[86] Mohammadreza Ghanavati, Diego Costa, Janos Seboek, David Lo, and Artur Andrzejak. Memory and resource leak defects and their repairs in java projects. *Empirical Software Engineering*, 25(1):678–718, 2020. URL: `http://dx.doi.org/10.1007/s10664-019-09731-8`, `doi:10.1007/s10664-019-09731-8`.

[87] Baljinder Ghotra, Shane McIntosh, and Ahmed E Hassan. Revisiting the impact of classification techniques on the performance of defect prediction models. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 789–800. IEEE, 2015. URL: `http://dx.doi.org/10.1109/ICSE.2015.91`, `doi:10.1109/ICSE.2015.91`.

[88] Emanuel Giger, Martin Pinzger, and Harald Gall. Predicting the fix time of bugs. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, pages 52–56, 2010.

[89] GitHub. Archiving repositories on github. URL: `https://docs.github.com/en/free-pro-team@latest/github/creating-cloning-and-archiving-repositories/about-archiving-repositories` [cited December 02, 2021].

[90] GitHub. Github rest api. URL: `https://docs.github.com/en/rest` [cited December 02, 2021].

[91] GitHut. Top active programming languages by githut. URL: `https://githut.info/` [cited December 02, 2021].

[92] Georgios Gousios. The ghtorent dataset and tool suite. In *Proceedings of the 10th working conference on mining software repositories*, pages 233–236. IEEE Press, 2013.

[93] Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, pages 345–355. ACM, 2014.

[94] Georgios Gousios and Diomidis Spinellis. Ghtorrent: Github's data from a firehose. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 12–21. IEEE, 2012. `doi:10.1109/MSR.2012.6224294`.

[95] Georgios Gousios and Diomidis Spinellis. Mining software engineering data from github. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 501–502. IEEE, 2017. URL: `http://dx.doi.org/10.1109/ICSE-C.2017.164`, `doi:10.1109/ICSE-C.2017.164`.

[96] Georgios Gousios, Margaret-Anne Storey, and Alberto Bacchelli. Work practices and challenges in pull-based development: The contributor's perspective. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 285–296. IEEE, 2016.

[97] Georgios Gousios, Bogdan Vasilescu, Alexander Serebrenik, and Andy Zaidman. Lean ghtorrent: Github data on demand. In *Proceedings of the 11th working conference on mining software repositories*, pages 384–387, 2014. URL: `http://dx.doi.org/10.1145/2597073.2597126`, `doi:10.1145/2597073.2597126`.

[98] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie Van Deursen. Work practices and challenges in pull-based development: the integrator's perspective. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 358–368. IEEE Press, 2015.

[99] Mark Grechanik, Chen Fu, and Qing Xie. Automatically finding performance problems with feedback-directed learning software testing. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 156–166. IEEE, 2012. URL: `http://dx.doi.org/10.1109/ICSE.2012.6227197`, `doi:10.1109/ICSE.2012.6227197`.

[100] Xiaohui Gu, Spiros Papadimitriou, S Yu Philip, and Shu-Ping Chang. Toward predictive failure management for distributed stream processing systems. In *2008 The 28th International Conference on Distributed Computing Systems*, pages 825–832. IEEE, 2008.

[101] Xiaohui Gu and Haixun Wang. Online anomaly prediction for robust cluster systems. In *2009 IEEE 25th International Conference on Data Engineering*, pages 1000–1011. IEEE, 2009.

[102] Qiang Guan and Song Fu. Wavelet-based multi-scale anomaly identification in cloud computing systems. In *2013 IEEE Global Communications Conference (GLOBECOM)*, pages 1379–1384. IEEE, 2013.

[103] Sumit Gulwani, Krishna K Mehra, and Trishul Chilimbi. Speed: precise and efficient static estimation of program computational complexity. *ACM Sigplan Notices*, 44(1):127–139, 2009. URL: `http://dx.doi.org/10.1145/1594834.1480898`, `doi:10.1145/1594834.1480898`.

[104] Zhen Guo, Guofei Jiang, Haifeng Chen, and Kenji Yoshihira. Tracking probabilistic correlation of monitoring data for fault detection in complex systems. In *International Conference on Dependable Systems and Networks (DSN'06)*, pages 259–268. IEEE, 2006.

[105] Shaifu Gupta, Neha Muthiyan, Siddhant Kumar, Aditya Nigam, and Dileep Aroor Dinesh. A supervised deep learning framework for proactive anomaly detection in cloud workloads. In *2017 14th IEEE India Council International Conference (INDICON)*, pages 1–6. IEEE, 2017.

[106] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. Performance debugging in the large via mining millions of stack traces. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 145–155. IEEE, 2012. URL: `http://dx.doi.org/10.1109/ICSE.2012.6227198`, `doi:10.1109/ICSE.2012.6227198`.

[107] Hapifhir. Commit c6777578 in project hapi-fhir. URL: `https://github.com/hapifhir/hapi-fhir/commit/c6777578a8b96eb5d9ea38bd280c40b8ca527e62` [cited May 07, 2022].

[108] Ahmed E Hassan. Predicting faults using the complexity of code changes. In *2009 IEEE 31st international conference on software engineering*, pages 78–88.

IEEE, 2009. URL: http://dx.doi.org/10.1109/ICSE.2009.5070510, doi:10.1109/ICSE.2009.5070510.

[109] Hazelcast. Issue 5104 in hazelcast. URL: https://github.com/hazelcast/hazelcast/issues/5104 [cited December 02, 2021].

[110] B2i Healthcare. Issue 626 in snow-owl. URL: https://github.com/b2ihealthcare/snow-owl/issues/626 [cited December 02, 2021].

[111] Steffen Herbold. Comments on scottknottesd in response to" an empirical comparison of model validation techniques for defect prediction models". *IEEE Transactions on Software Engineering*, 43(11):1091–1094, 2017. doi:10.1109/TSE.2017.2748129.

[112] Ralf Herbrich. Large margin rank boundaries for ordinal regression. *Advances in large margin classifiers*, pages 115–132, 2000.

[113] Ralf Herbrich, Thore Graepel, and Klaus Obermayer. Support vector learning for ordinal regression. 1999.

[114] Michiel Hermans and Benjamin Schrauwen. Training and analysing deep recurrent neural networks. In *Advances in neural information processing systems*, pages 190–198, 2013.

[115] Yoshiki Higo, Shinpei Hayashi, and Shinji Kusumoto. On tracking java methods with git mechanisms. *Journal of Systems and Software*, 165:110571, 2020.

[116] Abram Hindle, Daniel M German, and Ric Holt. What do large commits tell us?: a taxonomical study of large commits. In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 99–108. ACM, 2008.

[117] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[118] Pieter Hooimeijer and Westley Weimer. Modeling bug report quality. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 34–43. ACM, 2007.

[119] Tom Howley, Michael G Madden, Marie-Louise OConnell, and Alan G Ryder. The effect of principal component analysis on machine learning accuracy with high dimensional spectral data. In *International Conference on Innovative Techniques and Applications of Artificial Intelligence*, pages 209–222. Springer, 2005.

[120] Shaohan Huang, Carol Fung, Kui Wang, Polo Pei, Zhongzhi Luan, and Depei Qian. Using recurrent neural networks toward black-box system anomaly prediction. In *2016 IEEE/ACM 24th International Symposium on Quality of Service (IWQoS)*, pages 1–10. IEEE, 2016.

[121] Tian Huang, Yan Zhu, Qiannan Zhang, Yongxin Zhu, Dongyang Wang, Meikang Qiu, and Lei Liu. An lof-based adaptive anomaly detection scheme for cloud computing. In *2013 IEEE 37th Annual Computer Software and Applications Conference Workshops*, pages 206–211. IEEE, 2013.

[122] Xuan Huo, Yang Yang, Ming Li, and De-Chuan Zhan. Learning semantic features for software defect prediction by code comments embedding. In *2018 IEEE international conference on data mining (ICDM)*, pages 1049–1054. IEEE, 2018.

[123] Olumuyiwa Ibidunmoye, Ali-Reza Rezaie, and Erik Elmroth. Adaptive anomaly detection in performance metric streams. *IEEE Transactions on Network and Service Management*, 15(1):217–231, 2017.

[124] IBM. Ibm javametrics. URL: `https://developer.ibm.com/javasdk/application-metrics-java/` [cited December 2, 2019].

[125] IBM. Openliberty reference. URL: `https://openliberty.io/` [cited January 21, 2021].

[126] R Indhumathi and S Sathiyabama. Reducing and clustering high dimensional data through principal component analysis. *International Journal of Computer Applications*, 11(8):1–4, 2010.

[127] Glenn D Israel. Determining sample size. 1992. `doi:10.1177/104973200129118183`.

[128] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, 2016. URL: `https://aclanthology.org/P16-1195.pdf`, `doi:10.1109/ICSM.2013.38`.

[129] Enio G Jelihovschi, José Cláudio Faria, and Ivan Bezerra Allaman. Scottknott: a package for performing the scott-knott clustering algorithm in r. *TEMA (São Carlos)*, 15(1):3–17, 2014. URL: `https://www.scielo.br/j/tema/a/KMMPHsqnZnW9RnkmDdsYgtH/?lang=en`.

[130] Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. 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*, pages 111–120, 2009.

[131] Jfree. Candlestickrenderer.java in jfreechart. URL: `https://github.com/jfree/jfreechart/blob/master/src/main/java/org/jfree/chart/renderer/xy/CandlestickRenderer.java` [cited December 02, 2021].

[132] Guofei Jiang, Haifeng Chen, and Kenji Yoshihira. Discovering likely invariants of distributed transaction systems for autonomic system management. In *2006 IEEE International Conference on Autonomic Computing*, pages 199–208. IEEE, 2006.

[133] Guofei Jiang, Haifeng Chen, and Kenji Yoshihira. Modeling and tracking of transaction flow dynamics for fault detection in complex systems. *IEEE Transactions on Dependable and Secure Computing*, 3(4):312–326, 2006.

[134] Jing Jiang, David Lo, Jiateng Zheng, Xin Xia, Yun Yang, and Li Zhang. Who should make decision on this pull request? analyzing time-decaying relationships and file similarities for integrator prediction. *Journal of Systems and Software*, 2019.

[135] Jing Jiang, Yun Yang, Jiahuan He, Xavier Blanc, and Li Zhang. Who should comment on this pull request? analyzing attributes for more accurate commenter recommendation in pull-based development. *Information and Software Technology*, 84:48–62, 2017.

[136] Miao Jiang, Mohammad A Munawar, Thomas Reidemeister, and Paul AS Ward. Automatic fault detection and diagnosis in complex software systems by information-theoretic monitoring. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 285–294. IEEE, 2009.

[137] Miao Jiang, Mohammad A Munawar, Thomas Reidemeister, and Paul AS Ward. System monitoring with metric-correlation models: problems and solutions. In *Proceedings of the 6th international conference on Autonomic computing*, pages 13–22. ACM, 2009.

[138] Yujuan Jiang, Bram Adams, and Daniel M German. Will my patch make it? and how fast?: Case study on the linux kernel. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 101–110. IEEE Press, 2013.

[139] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices*, 47(6):77–88, 2012.

[140] Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. Catch me if you can: performance bug detection in the wild. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 155–170, 2011. URL: `http://dx.doi.org/10.1145/2048066.2048081`, `doi:10.1145/2048066.2048081`.

[141] JustGlowing. Minisom: a minimalistic implementation of the self organizing maps. URL: `https://github.com/JustGlowing/minisom` [cited November 27, 2019].

[142] Eirini Kalliamvakou, Daniela Damian, Leif Singer, and Daniel M German. The code-centric collaboration perspective: Evidence from github. *The Code-Centric Collaboration Perspective: Evidence from Github, Technical Report DCS-352-IR, University of Victoria*, page 17, 2014.

[143] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*, pages 92–101. ACM, 2014.

[144] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2012.

[145] Peter Kampstra. Beanplot: A boxplot alternative for visual comparison of distributions. *Journal of statistical software*, 28(1):1–9, 2008. `doi:10.18637/jss.v028.c01`.

[146] Eamonn Keogh, Jessica Lin, and Ada Fu. Hot sax: Efficiently finding the most unusual time series subsequence. In *Fifth IEEE International Conference on Data Mining (ICDM'05)*, pages 8–pp. Ieee, 2005.

[147] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, (1):41–50, 2003.

[148] Keras. Keras: The python deep learning library. URL: `https://keras.io/` [cited November 27, 2019].

[149] Charles Killian, Karthik Nagaraj, Salman Pervez, Ryan Braud, James W Anderson, and Ranjit Jhala. Finding latent performance bugs in systems implementations. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 17–26, 2010. URL: `http://dx.doi.org/10.1145/1882291.1882297`, `doi:10.1145/1882291.1882297`.

[150] Sunghun Kim, E James Whitehead, and Yi Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on software engineering*, 34(2):181–196, 2008.

[151] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[152] William H Kruskal and W Allen Wallis. Use of ranks in one-criterion variance analysis. *Journal of the American statistical Association*, 47(260):583–621, 1952.

[153] Divya Kumar and Krishn Kumar Mishra. The impacts of test automation on software's cost, quality and time to market. *Procedia Computer Science*, 79:8–15, 2016.

[154] Mayuresh Kunjir, Yuzhang Han, and Shivnath Babu. where does memory go?: Study of memory management in jvm-based data analytics . 2016.

[155] Christoph Laaber, Mikael Basmaci, and Pasquale Salza. Predicting unstable software benchmarks using static source code features. *Empirical Software Engineering*, 26(6):1–53, 2021. `doi:10.1007/s10664-021-09996-y`.

[156] Ahmed Lamkanfi and Serge Demeyer. Filtering bug reports for fix-time analysis. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 379–384. IEEE, 2012.

[157] Zhiling Lan, Ziming Zheng, and Yawei Li. Toward automated anomaly identification in large-scale systems. *IEEE Transactions on Parallel and Distributed Systems*, 21(2):174–187, 2009.

[158] Let's Dev Together (LDT). Abstractbuilding.java in project minecolonies. URL: `https://github.com/ldtteam/minecolonies/blob/version/main/src/main/java/com/minecolonies/coremod/colony/buildings/AbstractBuilding.java` [cited December 02, 2021].

[159] Let's Dev Together (LDT). Commit 508e7638 in project minecolonies. URL: `https://github.com/ldtteam/minecolonies/pull/6076/commits/508e763806951d74f4e8c2cafc87490d3a6d0ada` [cited December 02, 2021].

[160] Youngseok Lee, Suin Lee, Chan-Gun Lee, Ikjun Yeom, and Honguk Woo. Continual prediction of bug-fix time using deep learning-based activity stream embedding. *IEEE Access*, 8:10503–10515, 2020.

[161] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, 34(4):485–496, 2008. URL: `http://dx.doi.org/10.1109/TSE.2008.35`, `doi:10.1109/TSE.2008.35`.

[162] Christophe Leys, Christophe Ley, Olivier Klein, Philippe Bernard, and Laurent Licata. Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median. *Journal of Experimental Social Psychology*, 49(4):764–766, 2013.

[163] Hang Li. A short introduction to learning to rank. *IEICE TRANSACTIONS on Information and Systems*, 94(10):1854–1862, 2011.

[164] Hang Li. Learning to rank for information retrieval and natural language processing. *Synthesis Lectures on Human Language Technologies*, 7(3):1–121, 2014.

[165] Lexin Li. Dimension reduction for high-dimensional data. *Statistical methods in molecular biology*, pages 417–434, 2010.

[166] Zhixing Li, Gang Yin, Yue Yu, Tao Wang, and Huaimin Wang. Detecting duplicate pull-requests in github. In *Proceedings of the 9th Asia-Pacific Symposium on Internetware*, page 20. ACM, 2017.

[167] Hongliang Liang, Yue Yu, Lin Jiang, and Zhuosi Xie. Seml: A semantic lstm model for software defect prediction. *IEEE Access*, 7:83812–83824, 2019.

[168] Junhao Lin and Lu Lu. Semantic feature learning via dual sequences for defect prediction. *IEEE Access*, 9:13112–13124, 2021.

[169] Matteo Lissandrini, Martin Brugnara, and Yannis Velegrakis. Beyond macrobenchmarks: microbenchmark-based graph database evaluation. *Proceedings of the VLDB Endowment*, 12(4):390–403, 2018.

[170] Andrew G Liu, Ewa Musial, and Mei-Hwa Chen. Progressive reliability forecasting of service-oriented software. In *2011 IEEE International Conference on Web Services*, pages 532–539. IEEE, 2011.

[171] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th international conference on software engineering*, pages 1013–1024, 2014. doi:10.1145/2568225.2568229.

[172] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

[173] Pankaj Malhotra, Anusha Ramakrishnan, Gaurangi Anand, Lovekesh Vig, Puneet Agarwal, and Gautam Shroff. Lstm-based encoder-decoder for multi-sensor anomaly detection. *arXiv preprint arXiv:1607.00148*, 2016.

[174] Pankaj Malhotra, Lovekesh Vig, Gautam Shroff, and Puneet Agarwal. Long short term memory networks for anomaly detection in time series. In *Proceedings*, page 89. Presses universitaires de Louvain, 2015.

[175] Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947. URL: https://www.jstor.org/stable/2236101.

[176] Lionel Marks, Ying Zou, and Ahmed E Hassan. Studying the fix-time for bugs in large open source projects. In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, pages 1–8, 2011.

[177] Shigeru Maya, Ken Ueno, and Takeichiro Nishikawa. dlstm: a new approach for anomaly detection using deep learning with delayed prediction. *International Journal of Data Science and Analytics*, pages 1–28, 2019.

[178] Douglas R McCallum and James L Peterson. Computer-based readability indexes. In *Proceedings of the ACM'82 Conference*, pages 44–48. ACM, 1982.

[179] Mary L McHugh. Interrater reliability: the kappa statistic. *Biochemia medica*, 22(3):276–282, 2012. URL: `https://hrcak.srce.hr/89395`.

[180] Mekanism. Commit ee83cb14 in project mekanism. URL: `https://github.com/mekanism/Mekanism/commit/ee83cb142fc4c341750300a0e651cf79058a652b` [cited May 07, 2022].

[181] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE transactions on software engineering*, 33(1):2–13, 2006. URL: `http://dx.doi.org/10.1109/TSE.2007.256941`, `doi:10.1109/TSE.2007.256941`.

[182] Meta. Issue 127 in react-native. URL: `https://github.com/facebook/react-native/issues/127` [cited December 02, 2021].

[183] Donald Metzler and W Bruce Croft. Linear feature-based models for information retrieval. *Information Retrieval*, 10(3):257–274, 2007.

[184] Domas Mituzas. Embarrassment. *Blog post: Embarrassment*, 2009. URL: https://dom.as/2009/06/26/embarrassment/.

[185] I MOLYNEAUX. The art of application performance testing: Help for programmers and quality assurance, 2009.

[186] Glen Emerson Morris. lessons from the colorado benefits management system disaster. *Advertising and Marketing Review*, 2004. URL: https://www.academia.edu/1159972/UNMET_PUBLIC_EXPECTATIONS_PAY_IT_NOW_OR_PAY_IT_LATER_LESSONS_LEARNED_FROM_THE_COLORADO_BENEFITS_MANAGEMENT_SYSTEM_CBMS_.

[187] Mohammad A Munawar and Paul AS Ward. A comparative study of pairwise regression techniques for problem determination. In *Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*, pages 152–166. IBM Corp., 2007.

[188] Mohammad Ahmad Munawar and Paul AS Ward. Leveraging many simple statistical models to adaptively monitor software systems. In *International Symposium on Parallel and Distributed Processing and Applications*, pages 457–470. Springer, 2007.

[189] Kumiyo Nakakoji, Yasuhiro Yamamoto, Yoshiyuki Nishinaka, Kouichi Kishida, and Yunwen Ye. Evolution patterns of open-source software systems and communities. In *Proceedings of the international workshop on Principles of software evolution*, pages 76–85. ACM, 2002.

[190] PB Nemenyi. Distribution-free multiple comparisons [ph. d. dissertations]. *Princeton University*, 1963. URL: https://www.proquest.com/docview/302256074.

[191] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 902–912. IEEE, 2015. URL: http://dx.doi.org/10.1109/ICSE.2015.100, doi:10.1109/ICSE.2015.100.

[192] Adrian Nistor, Tian Jiang, and Lin Tan. Discovering, reporting, and fixing performance bugs. In *2013 10th working conference on mining software repositories (MSR)*, pages 237–246. IEEE, 2013. URL: http://dx.doi.org/10.1109/MSR.2013.6624035, doi:10.1109/MSR.2013.6624035.

[193] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 562–571. IEEE, 2013. URL: http://dx.doi.org/10.1109/ICSE.2013.6606602, doi:10.1109/ICSE.2013.6606602.

[194] Shuzi Niu, Jiafeng Guo, Yanyan Lan, and Xueqi Cheng. Top-k learning to rank: labeling, ranking and evaluation. In *Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval*, pages 751–760. ACM, 2012.

[195] Oswaldo Olivo, Isil Dillig, and Calvin Lin. Static detection of asymptotic performance bugs in collection traversals. In *Proceedings of the 36th ACM*

SIGPLAN Conference on Programming Language Design and Implementation, pages 369–378, 2015. URL: http://dx.doi.org/10.1145/2737924.2737966, doi:10.1145/2737924.2737966.

[196] Openjdk. Openjdk documentation. URL: http://openjdk.java.net/groups/hotspot/docs/HotSpotGlossary.html [cited December 2, 2019].

[197] OpenLiberty. Issue 226 in open-liberty. URL: https://github.com/OpenLiberty/open-liberty/issues/226 [cited December 02, 2021].

[198] Oracle. jconsole. URL: http://openjdk.java.net/tools/svc/jconsole/ [cited December 3, 2019].

[199] Rohan Padhye, Senthil Mani, and Vibha Singhal Sinha. A study of external community contribution to open-source projects on github. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 332–335. ACM, 2014.

[200] Husanbir S Pannu, Jianguo Liu, and Song Fu. A self-evolving anomaly detection framework for developing highly dependable utility clouds. In *2012 IEEE Global Communications Conference (GLOBECOM)*, pages 1605–1610. IEEE, 2012.

[201] Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013. URL: https://pragprog.com/titles/tpantlr2/the-definitive-antlr-4-reference.

[202] Terence Parr, Sam Harwell, and Kathleen Fisher. Adaptive ll (*) parsing: the power of dynamic analysis. *ACM SIGPLAN Notices*, 49(10):579–598,

2014. URL: http://dx.doi.org/10.1145/2714064.2660202, doi:10.1145/2714064.2660202.

[203] Raphael Pham, Leif Singer, Olga Liskin, Fernando Figueira Filho, and Kurt Schneider. Creating a shared understanding of testing culture on a social coding site. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 112–121. IEEE, 2013.

[204] Teerat Pitakrat, Dušan Okanović, André van Hoorn, and Lars Grunske. Hora: Architecture-aware online failure prediction. *Journal of Systems and Software*, 137:669–685, 2018.

[205] Thorsten Pohlert. The pairwise multiple comparison of mean ranks package (pmcmr). *R package*, 27(2019):9, 2014. URL: https://cran.microsoft.com/snapshot/2015-03-19/web/packages/PMCMR/vignettes/PMCMR.pdf3.

[206] Rob Powers, Moises Goldszmidt, and Ira Cohen. Short term performance forecasting in enterprise systems. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 801–807. ACM, 2005.

[207] Michael Pradel, Markus Huggler, and Thomas R Gross. Performance regression testing of concurrent classes. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 13–25, 2014. URL: http://dx.doi.org/10.1145/2610384.2610393, doi:10.1145/2610384.2610393.

[208] ProActive. Issue 2309 in scheduling. URL: https://github.com/ow2-proactive/scheduling/issues/2309 [cited December 02, 2021].

[209] The Netty Project. Issue 2384 in netty. URL: `https://github.com/netty/netty/issues/2384` [cited December 02, 2021].

[210] PYPL. Popularity of programming language. URL: `https://pypl.github.io/PYPL.html` [cited December 02, 2021].

[211] Mohammad Masudur Rahman, Chanchal K Roy, and Jason A Collins. Correct: code reviewer recommendation in github based on cross-project and technology experience. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 222–231. IEEE, 2016.

[212] Tim Richardson. census site still down after six months, 2002, 1901.

[213] Peter C Rigby, Daniel M German, and Margaret-Anne Storey. Open source software peer review practices: a case study of the apache server. In *Proceedings of the 30th international conference on Software engineering*, pages 541–550. ACM, 2008.

[214] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, and Jeff Skowronek. Appropriate statistics for ordinal level data: Should we really be using t-test and cohensd for evaluating group differences on the nsse and other surveys. In *annual meeting of the Florida Association of Institutional Research*, pages 1–33, 2006.

[215] Sudip Roy, Arnd Christian König, Igor Dvorkin, and Manish Kumar. Perfaugur: Robust diagnostics for performance anomalies in cloud services. In *2015 IEEE 31st International Conference on Data Engineering*, pages 1167–1178. IEEE, 2015.

[216] Ripon K Saha, Sarfraz Khurshid, and Dewayne E Perry. An empirical study of long lived bugs. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 144–153. IEEE, 2014.

[217] Sam Scott and Stan Matwin. Feature engineering for text classification. In *ICML*, volume 99, pages 379–388. Citeseer, 1999.

[218] Kai Shen, Christopher Stewart, Chuanpeng Li, and Xin Li. Reference-driven performance anomaly identification. In *ACM SIGMETRICS Performance Evaluation Review*, volume 37, pages 85–96. ACM, 2009.

[219] David J Sheskin. *Handbook of parametric and nonparametric statistical procedures*. crc Press, 2020. URL: `http://dx.doi.org/10.1201/9781420036268`, `doi:10.1201/9781420036268`.

[220] Shivkumar Shivaji, E James Whitehead, Ram Akella, and Sunghun Kim. Reducing features to improve code change-based bug prediction. *IEEE Transactions on Software Engineering*, 39(4):552–569, 2012.

[221] Leif Singer and Kurt Schneider. It was a bit of a race: Gamification of version control. In *2012 Second International Workshop on Games and Software Engineering: Realizing User Engagement with Game Engineering Techniques (GAS)*, pages 5–8. IEEE, 2012. URL: `http://dx.doi.org/10.1109/GAS.2012.6225927`, `doi:10.1109/GAS.2012.6225927`.

[222] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *ACM sigsoft software engineering notes*, 30(4):1–5, 2005. doi:10.1145/1082983.1083147.

[223] Derek Smith, Qiang Guan, and Song Fu. An anomaly detection framework for autonomic management of compute cloud systems. In *2010 IEEE 34th Annual Computer Software and Applications Conference Workshops*, pages 376–381. IEEE, 2010.

[224] Helen R Sofaer, Jennifer A Hoeting, and Catherine S Jarnevich. The area under the precision-recall curve as a performance metric for rare binary events. *Methods in Ecology and Evolution*, 10(4):565–577, 2019. doi:10.1111/2041-210X.13140.

[225] solarwinds. Solarwinds sam server & application monitor. URL: https://www.solarwinds.com/server-application-monitor?CMP=BIZ-TAD-PCWDLD-SAM_PP-A-PP-Q116 [cited November 27, 2019].

[226] Linhai Song and Shan Lu. Statistical debugging for real-world performance problems. *ACM SIGPLAN Notices*, 49(10):561–578, 2014.

[227] Linhai Song and Shan Lu. Performance diagnosis for inefficient loops. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 370–380. IEEE, 2017. URL: http://dx.doi.org/10.1109/ICSE.2017.41, doi:10.1109/ICSE.2017.41.

[228] Igor Steinmacher, Gustavo Pinto, Igor Scaliante Wiese, and Marco Aurélio Gerosa. Almost there: A study on quasi-contributors in open-source software

projects. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 256–266. IEEE, 2018.

[229] Christopher Stewart, Terence Kelly, and Alex Zhang. Exploiting nonstationarity for performance prediction. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 31–44. ACM, 2007.

[230] Jyothi Subramanian and Richard Simon. Overfitting in prediction models–is it a problem only in high dimensions? *Contemporary clinical trials*, 36(2):636–641, 2013.

[231] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. Lstm neural networks for language modeling. In *Thirteenth annual conference of the international speech communication association*, 2012.

[232] Seyyed Ehsan Salamati Taba, Foutse Khomh, Ying Zou, Ahmed E Hassan, and Meiyappan Nagappan. Predicting bugs using antipatterns. In *2013 IEEE International Conference on Software Maintenance*, pages 270–279. IEEE, 2013. URL: `http://dx.doi.org/10.1109/ICSM.2013.38`, `doi:10.1109/ICSM.2013.38`.

[233] Yongmin Tan et al. Online performance anomaly prediction and prevention for complex distributed systems. 2012.

[234] Yongmin Tan and Xiaohui Gu. On predictability of system anomalies in real world. In *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 133–140. IEEE, 2010.

[235] Yongmin Tan, Xiaohui Gu, and Haixun Wang. Adaptive system anomaly prediction for large-scale hosting infrastructures. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 173–182. ACM, 2010.

[236] Yongmin Tan, Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Chitra Venkatramani, and Deepak Rajan. Prepare: Predictive performance anomaly prevention for virtualized cloud systems. In *2012 IEEE 32nd International Conference on Distributed Computing Systems*, pages 285–294. IEEE, 2012.

[237] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering*, 43(1):1–18, 2016. URL: `http://dx.doi.org/10.1109/TSE.2016.2584050`, `doi:10.1109/TSE.2016.2584050`.

[238] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. The impact of automated parameter optimization on defect prediction models. *IEEE Transactions on Software Engineering*, 45(7):683–711, 2018. URL: `http://dx.doi.org/10.1109/TSE.2018.2794977`, `doi:10.1109/TSE.2018.2794977`.

[239] Adrian Taylor, Sylvain Apache Hadoop System. http://hadoop.apache.org/-core/.Leblanc, and Nathalie Japkowicz. Anomaly detection in automobile control network data with long short-term memory networks. In *2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pages 130–139. IEEE, 2016.

[240] Ted Tenny. Program readability: Procedures versus comments. *IEEE Transactions on Software Engineering*, 14(9):1271–1279, 1988.

[241] Patanamon Thongtanunam, Raula Gaikovina Kula, Ana Erika Camargo Cruz, Norihiro Yoshida, and Hajimu Iida. Improving code review effectiveness through reviewer recommendations. In *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*, pages 119–122. ACM, 2014.

[242] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken-ichi Matsumoto. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 141–150. IEEE, 2015.

[243] Tiobe. Tiobe index for popular programming languages. URL: `https://www.tiobe.com/tiobe-index/` [cited December 02, 2021].

[244] Apache Tomcat. Apache tomcat. URL: `https://tomcat.apache.org/` [cited January 15, 2021].

[245] Avishay Traeger, Ivan Deras, and Erez Zadok. Darc: Dynamic analysis of root causes of latency distributions. In *Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 277–288, 2008.

[246] Jason Tsay, Laura Dabbish, and James Herbsleb. Influence of social and technical factors for evaluating contribution in github. In *Proceedings of the 36th international conference on Software engineering*, pages 356–366. ACM, 2014.

[247] Dylan Tweney. Amazon website goes down for 40 minutes, costing the company $5 million, 2013. URL: `https://venturebeat.com/2013/08/19/amazon-website-down/` [cited November 26, 2019].

[248] Martijn van. Childrenquery file in commit 2d3df. URL: `https://github.com/elastic/elasticsearch/commit/2d3df921bcdb30ae607048d96650625a2ad2a904` [cited January 21, 2022].

[249] Martijn van. Commit 2d3df. URL: `https://github.com/elastic/elasticsearch/commit/2d3df921bcdb30ae607048d96650625a2ad2a904` [cited January 21, 2022].

[250] Erik Van Der Veen, Georgios Gousios, and Andy Zaidman. Automatically prioritizing pull requests. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 357–361. IEEE Press, 2015.

[251] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in github. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 805–816. ACM, 2015.

[252] VMware. Virtual machine cpu usage alarm. URL: `https://kb.vmware.com/s/article/2057830` [cited September 5, 2020].

[253] Chengwei Wang, Vanish Talwar, Karsten Schwan, and Parthasarathy Ranganathan. Online detection of utility cloud anomalies using metric distributions. In *2010 IEEE Network Operations and Management Symposium-NOMS 2010*, pages 96–103. IEEE, 2010.

[254] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 579–594. IEEE, 2017. URL: `http://dx.doi.org/10.1109/SP.2017.23`, `doi:10.1109/SP.2017.23`.

[255] Song Wang, Taiyue Liu, Jaechang Nam, and Lin Tan. Deep semantic feature learning for software defect prediction. *IEEE Transactions on Software Engineering*, 46(12):1267–1293, 2018.

[256] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 297–308. IEEE, 2016.

[257] Tao Wang, Jun Wei, Wenbo Zhang, Hua Zhong, and Tao Huang. Workload-aware anomaly detection for web applications. *Journal of Systems and Software*, 89:19–32, 2014.

[258] Yining Wang, Liwei Wang, Yuanzhi Li, Di He, Wei Chen, and Tie-Yan Liu. A theoretical analysis of ndcg ranking measures. In *Proceedings of the 26th Annual Conference on Learning Theory (COLT 2013)*, 2013.

[259] Jim / James C. Warner. top, linux man page, 2013. URL: `https://linux.die.net/man/1/top` [cited November 26, 2019].

[260] Webfx-project. Commit c241aa13 in project webfx. URL: `https://github.com/webfx-project/webfx/commit/c241aa134be0f744b213ab4380ff70a25dd25533` [cited May 07, 2022].

[261] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How long will it take to fix this bug? In *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*, pages 1–1. IEEE, 2007.

[262] Peter Weißgerber, Daniel Neu, and Stephan Diehl. Small patches get in! In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 67–76. ACM, 2008.

[263] Andrew W Williams, Soila M Pertet, and Priya Narasimhan. Tiresias: Black-box failure prediction in distributed systems. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–8. IEEE, 2007.

[264] Cort J Willmott and Kenji Matsuura. Advantages of the mean absolute error (mae) over the root mean square error (rmse) in assessing average model performance. *Climate research*, 30(1):79–82, 2005.

[265] Xin Xia, David Lo, Xinyu Wang, and Xiaohu Yang. Who should review this change?: Putting text and file location analyses together for more accurate recommendations. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 261–270. IEEE, 2015.

[266] Zhenglin Xia, Hailong Sun, Jing Jiang, Xu Wang, and Xudong Liu. A hybrid approach to code reviewer recommendation with collaborative filtering. In *2017*

6th International Workshop on Software Mining (SoftwareMining), pages 24–31. IEEE, 2017.

[267] Xusheng Xiao, Shi Han, Dongmei Zhang, and Tao Xie. Context-sensitive delta inference for identifying workload-dependent performance bottlenecks. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 90–100, 2013. URL: `http://dx.doi.org/10.1145/2483760.2483784`, `doi:10.1145/2483760.2483784`.

[268] XIAOHUI XIE. Principal component analysis. 2019.

[269] Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. Go with the flow: profiling copies to find runtime bloat. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 419–430, 2009. URL: `http://dx.doi.org/10.1145/1542476.1542523`, `doi:10.1145/1542476.1542523`.

[270] Guoqing Xu and Atanas Rountev. Detecting inefficiently-used containers to avoid bloat. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 160–173, 2010. URL: `http://dx.doi.org/10.1145/1806596.1806616`, `doi:10.1145/1806596.1806616`.

[271] Yue Yu, Huaimin Wang, Vladimir Filkov, Premkumar Devanbu, and Bogdan Vasilescu. Wait for it: Determinants of pull request evaluation latency on github. In *Mining software repositories (MSR), 2015 IEEE/ACM 12th working conference on*, pages 367–371. IEEE, 2015.

[272] Yue Yu, Huaimin Wang, Gang Yin, and Charles X Ling. Reviewer recommender of pull-requests in github. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 609–612. IEEE, 2014.

[273] Yue Yu, Huaimin Wang, Gang Yin, and Charles X Ling. Who should review this pull-request: Reviewer recommendation to expedite crowd collaboration. In *Software Engineering Conference (APSEC), 2014 21st Asia-Pacific*, volume 1, pages 335–342. IEEE, 2014.

[274] Yue Yu, Huaimin Wang, Gang Yin, and Tao Wang. Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment? *Information and Software Technology*, 74:204–218, 2016.

[275] Shahed Zaman, Bram Adams, and Ahmed E Hassan. Security versus performance bugs: a case study on firefox. In *Proceedings of the 8th working conference on mining software repositories*, pages 93–102, 2011. doi:10.1145/1985441.1985457.

[276] Shahed Zaman, Bram Adams, and Ahmed E Hassan. A qualitative study on performance bugs. In *2012 9th IEEE working conference on mining software repositories (MSR)*, pages 199–208. IEEE, 2012. URL: http://dx.doi.org/10.1109/MSR.2012.6224281, doi:10.1109/MSR.2012.6224281.

[277] Motahareh Bahrami Zanjani, Huzefa Kagdi, and Christian Bird. Automatically recommending peer reviewers in modern code review. *IEEE Transactions on Software Engineering*, 42(6):530–543, 2016.

[278] Dmitrijs Zaparanuks and Matthias Hauswirth. Algorithmic profiling. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 67–76, 2012. URL: http://dx.doi.org/10.1145/2254064.2254074, doi:10.1145/2254064.2254074.

[279] Jerrold H Zar. Spearman rank correlation. *Encyclopedia of Biostatistics*, 1998.

[280] Chen Zhang, Jiaxin Li, Dongsheng Li, and Xicheng Lu. Understanding and statically detecting synchronization performance bugs in distributed cloud systems. *IEEE Access*, 7:99123–99135, 2019. URL: http://dx.doi.org/10.1109/ACCESS.2019.2923956, doi:10.1109/ACCESS.2019.2923956.

[281] Feng Zhang, Ahmed E Hassan, Shane McIntosh, and Ying Zou. The use of summation to aggregate software metrics hinders the performance of defect prediction models. *IEEE Transactions on Software Engineering*, 43(5):476–491, 2016. URL: http://dx.doi.org/10.1109/TSE.2016.2599161, doi:10.1109/TSE.2016.2599161.

[282] Feng Zhang, Audris Mockus, Iman Keivanloo, and Ying Zou. Towards building a universal defect prediction model. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 182–191, 2014. URL: http://dx.doi.org/10.1145/2597073.2597078, doi:10.1145/2597073.2597078.

[283] Feng Zhang, Audris Mockus, Iman Keivanloo, and Ying Zou. Towards building a universal defect prediction model with rank transformed predictors. *Empirical Software Engineering*, 21(5):2107–2145, 2016. URL: http://dx.doi.org/10.1007/s10664-015-9396-2, doi:10.1007/s10664-015-9396-2.

[284] Feng Zhang, Quan Zheng, Ying Zou, and Ahmed E Hassan. Cross-project defect prediction using a connectivity-based unsupervised classifier. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 309–320. IEEE, 2016. URL: http://dx.doi.org/10.1145/2884781.2884839, doi:10.1145/2884781.2884839.

[285] Hongyu Zhang, Liang Gong, and Steve Versteeg. Predicting bug-fixing time: an empirical study of commercial software projects. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1042–1051. IEEE, 2013.

[286] Yutong Zhao, Lu Xiao, Xiao Wang, Lei Sun, Bihuan Chen, Yang Liu, and Andre B Bondi. How are performance issues caused and resolved?-an empirical study from a design perspective. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, pages 181–192, 2020. doi:10.1145/3358960.3379130.

[287] Chunting Zhou, Chonglin Sun, Zhiyuan Liu, and Francis Lau. A c-lstm neural network for text classification. *arXiv preprint arXiv:1511.08630*, 2015.

[288] Jian Zhou and Hongyu Zhang. Learning to rank duplicate bug reports. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 852–861. ACM, 2012.

[289] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*, pages 9–9. IEEE, 2007. URL: http://dx.doi.org/10.1109/PROMISE.2007.10, doi:10.1109/PROMISE.2007.10.