Thesis for The Degree of Doctor of Philosophy

Improving the Performance of Machine Learning-based Methods for Continuous Integration by Handling Noise

Khaled Walid Al-Sabbagh

Department of Computer Science and Engineering
Chalmers University of Technology | University of Gothenburg
Gothenburg, Sweden, 2023

Improving the Performance of Machine Learning-based Methods for Continuous Integration by Handling Noise

Khaled Walid Al-Sabbagh

ISBN: 978-91-8069-361-5 (PRINT)
ISBN: 978-91-8069-362-2 (PDF)

Department of Computer Science and Engineering
Division of Interaction Design and Software Engineering

Chalmers University of Technology | University of Gothenburg
SE-412 96 Göteborg,
Sweden
Phone: +46(0)729250522

Printed by Chalmers Digitaltryck,
Gothenburg, Sweden 2023.

"We are surrounded by data, but starved for insights."
- Jay Baer

Abstract

Background: Modern software development companies are increasingly implementing continuous integration (CI) practices to meet market demands for delivering high-quality features. The availability of data from CI systems presents an opportunity for these companies to leverage machine learning to create methods for optimizing the CI process.

Problem: The predictive performance of these methods can be hindered by inaccurate and irrelevant information – noise.

Objective: The goal of this thesis is to improve the effectiveness of machine learning-based methods for CI by handling noise in data extracted from source code.

Methods: This thesis employs design science research and controlled experiments to study the impact of noise-handling techniques in the context of CI. It involves developing ML-based methods for optimizing regression testing (MeBoTS and HiTTs), creating a taxonomy to reduce class noise, and implementing a class noise-handling technique (DB). Controlled experiments are carried out to examine the impact of class noise-handling on MeBoTS' performance for CI.

Results: The thesis findings show that handling class noise using the DB technique improves the performance of MeBoTS in test case selection and code change request predictions. The F1-score increases from 25% to 84% in test case selection and the Recall improved from 15% to 25% in code change request prediction after applying DB. However, handling attribute noise through a removal-based technique does not impact MeBoTS' performance, as the F1-score remains at 66%. For memory management and complexity code changes should be tested with performance, load, soak, stress, volume, and capacity tests. Additionally, using the "majority filter" algorithm improves MCC from 0.13 to 0.58 in build outcome prediction and from -0.03 to 0.57 in code change request prediction.

Conclusions: In conclusion, this thesis highlights the effectiveness of applying different class noise handling techniques to improve test case selection, build outcomes, and code change request predictions. Utilizing small code commits for training MeBoTS proves beneficial in filtering out test cases that do not reveal faults. Additionally, the taxonomy of dependencies offers an efficient and effective way for performing regression testing. Notably, handling attribute noise does not improve the predictions of test execution outcomes.

Keywords

Continuous Integration, Machine Learning, Class Noise, Attribute Noise.

List of Publications

Appended publications

This thesis is based on the following publications:

1. Al Sabbagh, K., Staron, M., Hebig, R., & Meding, W.(2019). Predicting Test Case Verdicts Using Textual Analysis of Committed Code Churns. In IWSM-Mensura. 2019, pp. 138–153
2. Al-Sabbagh, K. W., Hebig, R., & Staron, M.(2020). The effect of class noise on continuous test case selection: A controlled experiment on industrial data. In Product-Focused Software Process Improvement: 21st International Conference, PROFES 2020, Proceedings 21 (pp. 287-303)
3. Al Sabbagh, K., Staron, M., Hebig, R., & Meding, W.(2022). Improving test case selection by handling class and attribute noise.In Journal of Systems and Software, 183, 111093
4. Al-Sabbagh, K., Staron, M., Hebig, R., & Gomes, F.(2021). A classification of code changes and test types dependencies for improving machine learning based test selection. In Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering (pp. 40-49)
5. Al-Sabbagh, K. W., Staron, M., & Hebig, R.(2022). Improving Software Regression Testing Using a Machine Learning-Based Method for Test Type Selection. In Product-Focused Software Process Improvement: 23rd International Conference, PROFES 2022, Proceedings (pp. 480-496)
6. Al-Sabbagh, K. W., Staron, M., & Hebig, R.(2022). Predicting build outcomes in continuous integration using textual analysis of source code commits. In Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering (pp. 42-51)
7. Al-Sabbagh, K. W., Staron, M., & Hebig, R.(2023). The Impact of Class Noise Handling Techniques on the Effectiveness of Machine Learning-based Methods for Build Outcome and Negative Code Review Comment Predictions. Submitted to ACM Transactions on Software Engineering and Methodology

Other publications

The following publications were published before and during my PhD studies. However, they are not appended to this thesis, due to contents overlapping that of appended publications or contents not related to the thesis.

1. KW. Al-Sabbagh "Noise Handling For Improving Machine Learning-Based Test Case Selection"
   Licentiate thesis - Chalmers Library. (2021).

2. KW. Al-Sabbagh, M. Staron, M. Ochodek, R. Hebig, W. Meding "Selective Regression Testing based on Big Data: Comparing Feature Extraction Techniques"
   2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). IEEE. 2020, pp. 322–329.

3. KW. Al-Sabbagh, M. Staron, M. Ochodek, W. Meding "Early Prediction of Test Case Verdict withBag-of-Words vs. Word Embeddings"
   46th International Conference on Current Trends in Theory and Practice of Computer Science Workshops. (2020).

4. KW. Al-Sabbagh and L. Gren "The connections between group maturity, software development velocity, and planning effectiveness"
   Journal of Software: Evolution and Process, 30(1), p.e1896.

5. L. Gren and KW. Al-Sabbagh "Group Developmental Psychology and Software Development Performance "
   International Conference on Software Engineering Companion (ICSE-C). IEEE. 2017, pp. 232–234.

6. L. Bradley and KW. Al-Sabbagh "Mobile Language Learning Designs and Contexts for Newly Arrived Migrants." *Australian Journal of Applied Linguistics 5, no. 3 (2022): 179-198.*

7. L. Bradley, L. Bartram, KW. Al-Sabbagh, A. Algers "Designing mobile language learning with Arabic speaking migrants "
   Interactive Learning Environments, pp.1-13. 2020.

8. KW. Al-Sabbagh, L. Bradley, L. Bartram "Mobile language learning applications for Arabic speaking migrants – a usability perspective "
   Language Learning in Higher Education9.1 (2019), pp. 71–95.

9. L. Bartram, Lorna, L. Bradley, and KW. Al-Sabbagh. "Mobile learning with Arabic speakers in Sweden."
   In Proceedings of the Gulf Comparative Education Symposium (GCES) in Ras Al Khaimah, UAE, pp. 5-11. 2018.

Research Contribution

In this thesis, the CRediT (Contribution Roles Taxonomy) model [1] was employed to specify the authors' contributions to the publications included in this thesis. Table 1 provides an overview of these contributions.

Table 1: The authors' contributions to the appended papers that comprise this thesis.

Role	Khaled Sabbagh	Miroslaw Staron	Regina Hebig	Wilhelm Meding	Francisco Gomes
Conceptual-ization	Papers B, C, D, E, F, G	Papers A, B, C, D, G	Papers B, G	Paper A	-
Methodology	Papers B, C, D, E, F, G	Papers A, B, C, D, G	Papers A, B, G	-	-
Software	Papers A, B, C, E, F, G	Papers A, C	Paper A	-	-
Validation	Papers A, B, C, D, E, F, G	Papers A, B, G	Paper A, G	-	-
Formal analysis	Papers A, B, C, D, E, F, G	Papers A, B, D	Paper A	-	-
Investigation	Papers A, B, C, D, E, F, G	A, B	A	-	-
Resources	Papers A, B, C, D, E, F, G	Papers B, C, D, E, F, G	Paper D	-	-
Data curation	Papers A, B, C, D, E, F, G	-	-	-	-
Writing original draft	Papers A, B, C, D, E, F, G	Papers A, B, D, G	Papers A, B, F	Paper A	Paper D
Writing review & editing	Papers A, B, C, D, E, F, G	Papers A, B, C, D, E, F, G	Papers A, B, C, E, F, G	Paper C	Paper D
Visualization	Papers A, B, C, D, E, F, G	Papers A, B, C, D, G	Paper A	-	Paper D
Supervision	-	Papers A, B, C, D, E, F, G	Papers A, B, C, D, E, F, G	Paper A	-
Project admin-istration	Papers A, B, C, D, E, F, G	Papers A, B, C, D, E, F, G	Papers A, B, C, D, E, F, G	Paper A	Paper D
Funding acquisition	-	Papers A, B, C, D, E, F, G		Papers A, B, C	-

Acknowledgment

Of all the gifts that I received as a Ph.D candidate, the greatest, undoubtedly, has been the people I met and worked with. Foremost, I would like to thank my academic advisors, Miroslaw Staron and Regina Hebig, for their valuable guidance and stimulating research discussions. Both Miroslaw and Regina have brought structure into my research work and contributed significantly to my growth as a researcher. My gratitude is also extended to my examiner, Jan Bosch, whose feedback has instilled discipline and rigor into my work.

Special gratitude is owed to several individuals from Software Center, especially to Wilhelm Meding, whose support and continuous advice were of great help. I also extend my appreciation to the software engineers from Deif, Axis, Ericsson, and Grundfos, who participated in my research studies and offered guidance when needed.

I am very grateful to Francisco Gomes, whose wisdom and support have been instrumental to help me navigate several challenges during my Ph.D. My heartfelt appreciation extends to Lucas Gren for all the profound discussions we shared, and the moments of spontaneous laughter. I hold a deep sense of gratitude for my friendship with Mazen Mohammad, who has always been reliable and kind. I would like to extend my gratitude to every individual in the division of interaction design and software engineering. Your presence has been a wellspring of inspiration.

Special appreciation goes to my friends who have been pillars of support throughout my journey: To Alaa Alnuweiri, for his steadfast companionship, standing beside me through both the highs and lows. To Linda Bradley and her family, for their constant support, belief in me, and encouragement. To Abiya Touma for her unconditional kindness. I would also like to thank Peter Samoaa. I cherish the time we shared during different phases of my Ph.D.

My gratitude is extended to my loving family in Syria, without whom nothing would have been possible. In particular, I want to thank my mother, sisters, and brother whose sacrifices have played a pivotal role in my professional growth. I would also like to acknowledge my uncle Haitham, whose guidance has greatly influenced the person I have become today.

A special thank goes to my amazing wife Joumana, who has been a constant source of strength and inspiration, particularly during challenging times. I am deeply grateful for all the boundless warmth and love that she has shown.

Contents

Abstract iii

List of Publications v

Acknowledgement ix

1 Introduction 1

1.1 Theoretical Framework 4

1.1.1 Continuous Integration 4

1.1.2 Test Case Selection 5

1.1.3 Noise data types: class and attribute noise 5

1.1.4 Sources of class and attribute noise in CI 6

1.1.5 Noise-handling strategies 9

1.2 Related Work 11

1.2.1 Machine learning-based approaches in continuous integration 12

1.2.2 Noise-handling in software engineering contexts 13

1.3 Research Focus and Questions 14

1.4 Research Methodology 18

1.4.1 Design Science Research 19

1.4.2 Controlled Experiments 19

1.4.3 Research Methods 20

1.5 Summary of the Findings 26

1.6 Research Validity 30

1.6.1 External Validity 30

1.6.2 Internal Validity 31

1.6.3 Construct Validity 31

1.6.4 Conclusion Validity 32

1.7 Discussion 32

1.8 Conclusion and Future Work 34

2 Paper A 37

2.1 Introduction 37

2.2 Background 38

2.2.1 Categories of Machine Learning 38

2.2.2 Tree-based and Deep Learning Models 38

2.2.3 Code Churns 39
2.3 Related Work 39
2.3.1 ML-based Test-Case Selection 39
2.4 Method using Bag of Words for Test Selection (MeBoTS) . . . 40
2.4.1 Code Churns Extraction (Step 1) 40
2.4.2 Textual Analysis and Features Extraction (Step 2) . . . 41
2.4.3 Training and Applying the Classifier Algorithm (Step 3) 43
2.5 Research Design 43
2.5.1 Collaborating Company 43
2.5.2 Dataset 43
2.5.3 Evaluating and Selecting a Classification Model 45
2.6 Results 47
2.6.1 Training the Models on Churns of Varying Sizes 47
2.6.2 Training the Models on Churns of Small Sizes 47
2.6.3 Implication 48
2.7 Validity analysis 48
2.8 Recommendations 49
2.9 Conclusion and Future Work 50

3 Paper B 51
3.1 Introduction 51
3.2 Definition and Example of class Noise in Source Code 52
3.3 Related Work 54
3.3.1 The Impact of Noise on Classification Performance . . . 54
3.3.2 Text Mining for Test Case Selection and Defect Prediction 55
3.4 Experiment Design 55
3.4.1 Data Collection Method 55
3.4.2 Independent Variable and Experimental Subjects 56
3.4.3 Dependent Variables 56
3.4.4 Experimental Hypotheses 57
3.4.5 Data Analysis Methods 57
3.5 Experiment Operations 57
3.5.1 Creation of The Control Group 57
3.5.2 Class Noise Generation 59
3.5.3 Performance Evaluation Using Random Forest 60
3.6 Results 60
3.6.1 Descriptive Statistics 60
3.6.2 Hypotheses Testing 61
3.7 Threats to Validity 63
3.8 Conclusion and Future Work 65

4 Paper C 67
4.1 Introduction 67
4.2 Related Work 69
4.2.1 Text Mining For Defect Prediction and Test Case Selection 70
4.2.2 Class Noise Handling Research 71

4.2.3 Attribute Noise Handling Research 72
4.3 Background, Definitions, and Examples 73
4.3.1 Core Concepts 73
4.3.2 Method Using Bag of Words For Test Case Selection (MeBoTS) 74
4.3.3 Noise Definitions and Examples 76
4.3.3.1 Example of the dependency between code churns and test case verdict 76
4.3.3.2 Definition and Example of Class Noise in Code Churns Data 76
4.3.4 Definition and Example of Attribute Noise in Code Churns Data 78
4.4 Noise Handling and Removal Approaches 79
4.4.1 Class Noise Approach 80
4.4.2 Selected Attribute Noise Handling Approach 81
4.5 Research Methodology 81
4.5.1 Original Data Set 82
4.5.2 Random Forest For Evaluation 83
4.5.3 Class Noise 83
4.5.4 Attribute Noise 83
4.5.4.1 Adopted Data-Set 84
4.5.4.2 Independent Variable and Experimental Subjects 84
4.5.4.3 Dependent Variables 85
4.5.4.4 Experimental Hypotheses 85
4.5.4.5 Data Analysis Methods 86
4.5.4.6 Attribute Noise Removal 86
4.6 Evaluation Results 87
4.6.1 Original vs. Class Noise Cleaned Data 87
4.6.2 Class Noise Cleaned vs. Class and Attribute Noise Cleaned Data 87
4.7 Discussion 93
4.8 Threats to Validity 95
4.9 Conclusion and Future Work 97
4.10 Appendix A 98

5 Paper D 99
5.1 Introduction 99
5.2 Related Work 100
5.2.1 Defect Taxonomies 100
5.2.2 Taxonomies in Software Testing 100
5.3 Research Method 101
5.3.1 Planning 101
5.3.2 Identification and Extraction 101
5.3.3 Design and Construction 102
5.3.3.1 Survey 103
5.3.3.2 Workshop with Testers 103
5.3.4 Validation 104

5.4 Results 105
5.4.1 Test Case Types 105
5.4.2 Code Change Categories and Dependencies with Test Case Types 105
5.4.3 Dependency Patterns and Strengths 109
5.4.3.1 Survey. 109
5.4.3.2 Workshop 110
5.4.3.3 Memory Management 110
5.4.3.4 Complexity code changes 114
5.4.4 Resulting Taxonomy 115
5.5 Taxonomy Validation 116
5.5.1 Orthogonality of the Taxonomy's Facets 116
5.5.2 Instrumenting Prediction of Dependencies 116
5.6 Threats to validity 118
5.7 Conclusion and Future Work 119

6 Paper E 121
6.1 Introduction 121
6.2 Related Work 122
6.3 Core Concepts and Background 123
6.3.1 Core Concepts 123
6.3.2 The Dependency Taxonomy 123
6.4 Research Design 124
6.4.1 HiTTs Implementation 125
6.4.1.1 Annotation and Training (Phase 1): 125
6.4.1.2 Calibration (Phase 2): 126
6.4.1.3 Selection (Phase 3): 126
6.4.2 Usage Scenario 127
6.5 Evaluation of HiTTs 127
6.5.1 Annotation and Training (Phase 1) 127
6.5.1.1 Code Change Extraction (step 1) 128
6.5.1.2 Annotation and Class Balancing (step 2) . . . 128
6.5.1.3 Features Extraction and Building the Classifier (step 3) 129
6.5.2 Calibration (Phase 2) 129
6.5.3 Selection (Phase 3) 130
6.5.4 Baseline Construction 130
6.5.4.1 Actual 131
6.5.4.2 Retest-all 131
6.5.5 Results and Analysis 132
6.6 Threats to Validity 133
6.7 Conclusion and Future Work 135

7 Paper F 137
7.1 Introduction 137
7.2 Related Work 138
7.2.1 Software Metrics for Build Prediction 139
7.2.2 Reasons of Build Failure in CI 139
7.3 Experiment Design and Operations 140
7.3.1 Data Collection and Preprocessing 140
7.3.1.1 Traditional Software Metrics 140
7.3.1.2 The Token Frequency Metric 140
7.3.2 Independent Variables 143
7.3.3 Evaluation Metrics 144
7.3.4 Experimental Hypotheses 144
7.3.5 Data Analysis Methods 145
7.3.6 Prediction Model 145
7.3.7 Experimental Subjects and Class Balancing 146
7.4 Results 146
7.4.1 Evaluation of Metrics effectiveness 146
7.4.2 Hypotheses Testing 148
7.4.2.1 Significance Testing 148
7.4.2.2 Effect Size 151
7.5 Threats to Validity 152
7.6 Conclusion and Future work 153

8 Paper G 155
8.1 Introduction 155
8.2 Related Work 157
8.2.1 Removal and correction based techniques for class noise-handling 157
8.2.2 Class noise-handling in software engineering contexts . . 159
8.3 Background 161
8.3.1 Definition and example of class noise 161
8.3.2 Code review process in Gerrit 161
8.3.3 Class noise example in code review data 162
8.3.4 Noise-handling techniques 163
8.3.4.1 Removal-based noise-handling techniques . . . 163
8.3.4.2 Domain knowledge-based noise-handling technique (DB) 164
8.4 Research Design 165
8.4.1 Data collection and preparation (Part one) 165
8.4.1.1 Experiment variables 165
8.4.1.2 Collection of build outcomes data (Experiment 1) 166
8.4.1.3 Collection of code reviews data (Experiment 2) 167
8.4.1.4 Feature extraction 170
8.4.2 Design of the experiments 170
8.4.2.1 Cross validation 171
8.4.2.2 Measurement method 171
8.4.2.3 Individual improvement 172

8.4.2.4 Effect size 172
8.4.2.5 Significance testing 172
8.5 Results 173
8.5.1 The impact of class noise-handling on predicting the outcome of builds in continuous integration (RQ1) . . . 173
8.5.1.1 Individual improvements 174
8.5.1.2 Descriptive statistics 177
8.5.1.3 Hypotheses testing 180
8.5.2 The impact of class noise-handling on predicting code change requests (RQ2) 181
8.5.2.1 Descriptive statistics 182
8.5.3 Hypotheses testing 184
8.6 Discussion 186
8.6.1 RQ1- What is the impact of applying class noise-handling techniques on predicting the outcome of builds in continuous integration? 186
8.6.2 RQ2: What is the impact of applying class noise-handling techniques on predicting code change requests? 187
8.6.3 Characteristics of noisy lines of code 187
8.6.4 Confounding factors 188
8.7 Research Validity 189
8.7.1 External validity threats 189
8.7.2 Internal validity threats 190
8.7.3 Construct validity threats 190
8.7.4 Conclusion validity threats 190
8.8 Conclusion and Future Work 191
8.9 Appendix A 193

Bibliography **205**

Chapter 1

Introduction

Modern software development companies need to keep up with the ever-growing market demands for delivering complex and high-quality features at lower costs. To meet these challenges, companies adopt the practice of continuous integration (CI) and create automated tools to optimize their CI process. Adopting CI offers companies the benefits of continuously verifying the integrity of code changes at frequent intervals, which allows for early detection of faults, rapid feedback to developers, and improved collaborations among team members [2].

The CI process typically comprises a series of steps. These include building and testing code changes committed by software engineers to a version control system. The building step includes tasks such as compiling new features, resolving dependencies, and creating executable artifacts. Once the code is transformed into an executable form, it undergoes a phase of testing, where automated test cases are executed to identify whether new faults have been introduced into the code. After completing the building and testing steps, software engineers receive feedback from the CI system regarding the status of their committed code. This feedback informs engineers about whether their code has successfully passed the CI steps or requires further scrutiny and bug-fixing.

While CI offers advantages that accelerate feature delivery, organizations that adopt CI face the challenge of reducing the latency in feedback between CI and software engineers without compromising the effectiveness of fault detection. With increased code integration frequency and complexity of features in source-code files, it becomes important to develop tools that can optimize the effectiveness of the CI process such that fault-prone code changes are identified and reported to software engineers as early as possible.

The availability of large amounts of data from CI systems presents researchers and practitioners with an opportunity to develop data-driven approaches that can optimize the automation of tools in CI. As a result, a multitude of research studies have been conducted to investigate the use of machine learning (ML) for optimizing tools' automation within the CI process. For example, Hassan and Zhang [3] conducted a study in which they mined a diverse set of product and process metrics from historical projects. These metrics included the number of modified subsystems and certification results of previous builds. They utilized this data to construct an ML model for build prediction. The results of their study demonstrated that training a decision tree classifier with such information achieved a correct prediction rate of 69% for failing builds.

Similarly, Xia and Li [4] performed an evaluation involving nine classifiers and 20 software metrics for 126 open-source projects. Their findings revealed that using these metrics led to an F1-score exceeding 70% for 21 build outcomes. These results indicate that product and process metrics hold promise in predicting build outcomes effectively. These approaches employ a classifier that gets trained on both faulty and non-faulty code examples to predict whether new code commits will successfully build and pass the testing phase in the CI pipeline. These approaches have demonstrated their effectiveness in solving CI-specific tasks, including test case selection, build outcome predictions, bug-fix time estimations, and more.

While ML-based approaches have demonstrated promising potential in the context of CI, their utility can be hindered by the presence of noise in the training data. This noise refers to inaccurate and irrelevant information in the training entries of a given data-set [5]. Two categories of noise commonly discussed in the literature are class and attribute noise [6]. Class noise arises from contradictory or mislabelled instances in the training data, while attribute noise occurs when attributes contain irrelevant or missing information [7].

In the context of CI, we can observe class noise in code changes that are assigned with incorrect class labels or code changes that appear multiple times with inconsistent class labels (i.e., contradictory). On the other hand, attribute noise can be observed when irrelevant or missing information within the attributes or features is used to describe the code changes. This inaccurate information in the class and attribute values makes it difficult for ML models to learn patterns about faulty code changes.

To address the problem of noise, researchers proposed strategies that can be used for handling the effect of noise. These strategies can be broadly classified into three categories: tolerance, elimination, and correction [6]. The tolerance-based category deals with noise by leaving it in place and, instead, relies on designing robust ML techniques that can tolerate noise to a certain threshold. The removal-based category seeks to identify instances with class noise and then removes them from the data-set. Finally, techniques in the correction category seek to correct mislabeled entries by replacing their values with ones that are more appropriate.

Although these noise-handling strategies have been extensively studied in the field of machine learning, their application and impact within the context of CI have not been examined. CI differs from other contexts in the way code-change data gets continuously and frequently pushed, built, and tested. Due to that difference, it cannot be assumed that the impact of noise-handling on ML-based methods for CI is similar to those reported in the literature in different contexts. Thus, it is important to examine the impact of noise in code changes collected during CI.

Noise in CI can arise due to several factors. One such factor is the accuracy of the measurement instruments used to measure code metrics. If the measurement tool does not accurately measure what it claims to measure, then attribute noise can be introduced. For example, if a static analysis tool inaccurately measures the McCabe complexity of a program, it introduces inaccurate information about the code complexity, and thus attribute noise.

Another factor that introduces noise is the presence of flaky tests. Flaky tests can produce execution results that falsely indicate faults or non-faults in code changes. When using their execution outcomes as class labels for code changes, these flaky tests contribute to the introduction of class noise. Similarly, tests and build can sometimes be interrupted when they get executed during an environment upgrade. In such situations, the execution of tests or builds may be disrupted, leading to incorrect class values assigned to code changes.

Additionally, the labeling mechanism of code changes can be another source of class noise. In scenarios like test case selection, where faulty lines of code are unknown, a common practice is to label all lines of code in a code commit with the execution result of a test case. However, such a labeling mechanism introduces class noise, as not all lines of code within commits are faulty and relevant to the observed test execution result used for labeling.

The main goal of this thesis is to improve the effectiveness of ML-based tools in the context of CI by handling class and attribute noise in CI data. To achieve our goal, we conducted a series of design science research and controlled experiments studies. In the design science studies, we developed and evaluated the effectiveness of an ML-based tool (MeBoTS) to assist software engineers optimize regression testing. Thereafter, we created a taxonomy of dependency between code changes and test case types to reduce class noise in the training data of code changes. To validate the taxonomy, we developed another method (HiTTs) that classifies code changes into different categories and selectively executes test cases based on the most occurring code changes. We then conducted controlled experiments to investigate the effects of noise and different noise-handling strategies on the effectiveness of MeBoTS in build outcome predictions, test case selection, and negative code review comments predictions.

This thesis consists of this introduction chapter and seven other chapters, each based on a research paper. The introduction chapter is structured as follows: In Section 1.1, we introduce and describe the theory that explains why the research problem presented in this thesis is questioned. In Section 1.2, we highlight related work that concerns ML-based approaches in CI and existing noise-handling techniques in the software engineering literature. In Section 1.3, we outline the general research question that guided the conduct of the included research studies. The methodology employed to obtain our results is detailed in Section 1.4. Section 1.5 outlines the findings and contributions of this thesis. In Section 1.6, we discuss the threats to the validity of the appended papers. The answer to the general research question is provided in Section 1.7. Finally, Section 1.8 provides a summary of our conclusions and discuss potential avenues for future work.

1.1 Theoretical Framework

In this section, we provide an overview of the fundamental concepts and code examples that are essential for comprehending the content of this thesis. We begin by describing the practice of continuous integration and the process it incorporates. After that, we describe how noise can be introduced in CI and illustrate different types of noise-handling strategies that we analyze for an impact on ML-based methods in the context of CI.

1.1.1 Continuous Integration

Continuous Integration is a software development practice that focuses on frequently integrating code changes that get tested by an automated build system [8]. This frequent integration and testing performed by CI servers allow software engineers to detect faults early before new faults propagate into the code-base. As a result, CI reduces the burden and effort of tracking faults after they have propagated into other components of the system under test. A CI process typically consists of three sequential steps that automate the building and testing of code changes. Figure 1.1 illustrates each of these steps.

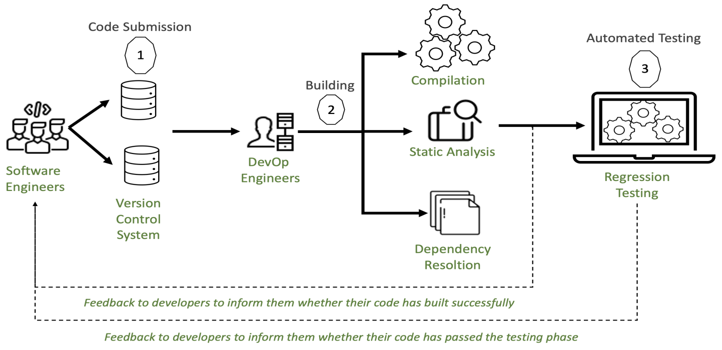

Code Submission
1
Software Engineers
Version Control System
DevOp Engineers
Building
2
Compilation
Static Analysis
Dependency Resoltion
Automated Testing
3
Regression Testing
Feedback to developers to inform them whether their code has built successfully
Feedback to developers to inform them whether their code has passed the testing phase

Figure 1.1: The continuous integration process.

The first step in the CI workflow is code submission, where developers work on their individual branches or forks to submit changes and add new features to the code-base. The code submission step involves committing new code changes into a shared repository hosted by a version control system, allowing different code branches to be merged and built.

The second step is automated building. This step involves automatically compiling the code, resolving dependencies, and generating an executable artifact. The goal of this step is to ensure that the code-base can be successfully transformed into an executable form, catching any compilation errors, missing dependencies, or coding style violations [9].

The third step in the CI process is automated testing. Once the code changes have been successfully built and an executable artifact has been generated, a suite of test cases is executed to ensure that previously implemented functionality continues to work as expected after new code changes or system modifications are made [10]. This type of testing is known as regression testing.

1.1.2 Test Case Selection

Regression testing ensures that previously implemented functionality continues to work as expected after new code changes or system modifications are made [10]. However, regression testing can be time-consuming and resource-intensive, since all test cases available in the suite of tests get executed at regular intervals. To address this challenge, techniques like test case selection are employed to optimize the regression testing process.

Test case selection is a technique that aims at reducing the time of regression testing by identifying a subset of test cases that effectively exercises parts of the system that have been affected by code changes. By selecting these relevant test cases, the testing effort can be focused on parts of the system that are more likely to be impacted by the modifications, thereby saving time and resources. This type of technique is leveraged by the CI server for testing frequently submitted code changes immediately after every successful build.

In order to increase confidence in the testing of the system, a daily suite is scheduled to run overnight, encompassing a more comprehensive set of tests that cover a wider range of functionalities and cases. These tests help identify any issues that may have been missed during the initial regression testing after the build. Lastly, the weekly suite is executed over the weekend and encompasses an even broader set of tests. These tests aim to validate the overall system behavior, ensuring the system's compliance with various requirements and specifications.

Figure 1.2 exemplifies how companies perform regression testing by organizing three types of suites. The *every build* suite in the figure comprises a subset of test cases that are deemed effective in revealing faults given the new code changes. The desired outcome from the utilized technique is to reveal all faults immediately after a successful build, and hence save developers time and effort that would otherwise be required to address these faults after they have spread to other parts of the system.

Therefore, an effective test case selection technique is determined by its ability to detect faults after every build, such that no new faults are detected when executing the daily and weekly suites.

1.1.3 Noise data types: class and attribute noise

The quality of real-world data is inherently imperfect since it contains a large amount of entries that come with corrupted information. Those entries can adversely affect the performance of classifiers in performing their designated prediction tasks [11].

Among the components that determine the quality of data is the accuracy of the information within the class and attribute values. The accuracy of class values is determined by whether the class of training entries is correctly assigned. The accuracy of attributes is determined by whether the attribute

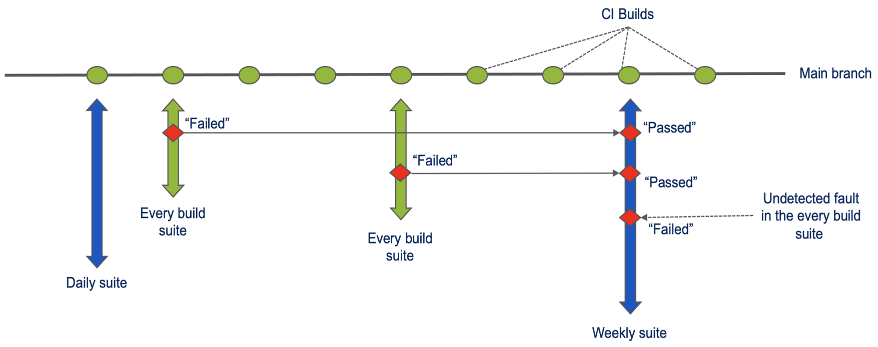


Figure 1.2: An illustration of regression testing in CI.

values correctly characterize the training entries for classification. Based on these two distinctions between the quality of class and attribute values, the following two types of noise can be identified in data-sets in general, including CI data:

- Class noise: it occurs when a training entry is incorrectly labeled. Two types of class noise can be distinguished:
  - Contradictory entries: these are identical entries in the data having different class values.
  - Misclassification: these are training entries in the data that are labeled with class values different from their true values.
- Attribute noise: it occurs when one or more attribute values of a training entry are erroneous, missing, or deviate substantially from the majority of entries.

1.1.4 Sources of class and attribute noise in CI

We now turn to discuss the sources of class and attribute noise in CI data. Figure 1.3 presents a taxonomy outlining eight sources of noise that we identified in the context of CI.

Sources of class noise In the context of CI, class noise can occur when the class values assigned to individual lines of code are inaccurate. This inaccuracy can be observed when identical lines of code are assigned different class values and when lines of code are misclassified. In the context of CI, we identified six potential sources of class noise:

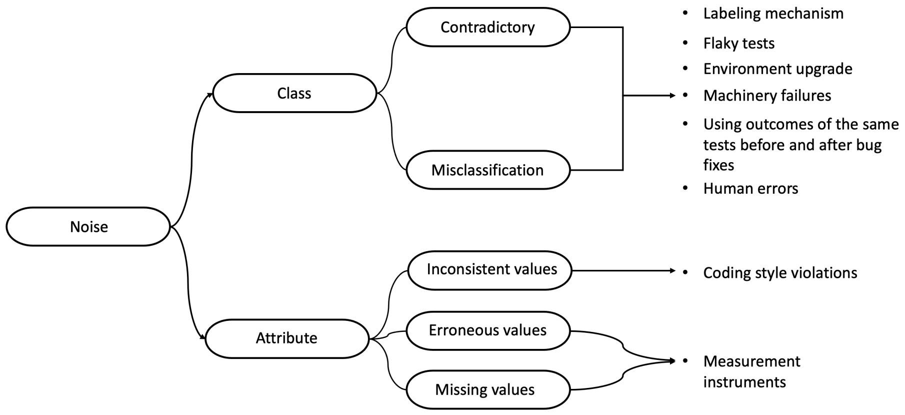


Figure 1.3: An overview of noise types and their sources in CI.

1) labeling mechanism: The first source of class noise is the labeling mechanism in which class values are extracted from databases and assigned to individual lines of code. In some cases, a labeling mechanism might rely on information stored in databases, such as the execution outcome of a test case, to determine the class value of lines of code within a particular commit.

Consider a scenario where a labeling mechanism assigns the execution outcome of a specific test case *tc1* that passes the execution as the class value to all lines of code in a given commit. Now, imagine that a fraction of the same lines of code appear in another commit where *tc1* 'fail' and reveals a fault in the commit. In such a scenario, contradictory entries arise because the labeling mechanism assigns inconsistent class values to the same lines of code across multiple commits. As a result, the same line of code may be observed multiple times with "pass" and "fail" class values.

2) flaky tests: Flaky tests present another potential source of class noise. These tests exhibit inconsistent behavior, meaning they can produce both passing and failing results when executed against the same version of the source code [12]. Since flaky tests can yield different outcomes across multiple test runs, assigning their execution outcomes as class values to individual lines of code can result in contradictory entries. For example, a line of code may be labeled as "pass" in one test run where the flaky test succeeds, but labeled as "fail" in another test run where the same flaky test "fail".

3) environment upgrade: Environment upgrades introduce an additional source of class noise. In a CI pipeline, a regular upgrade to the software environment in which the CI pipeline operates is performed. This typically involves upgrading the infrastructure, tools, frameworks, etc that are utilized within the CI pipeline. During an environment upgrade, the CI server may need to be temporarily taken offline or restarted several times. This downtime can disrupt ongoing build jobs or tests amidst their execution, causing their execution outcomes to fail. As a consequence, assigning the execution outcome of builds or tests that got disrupted during an environment upgrade can result in introducing contradictory entries. For example, a line of code may be labeled as “fail” in one build or test run due to a disruption caused by an environment upgrade. In another run, the same line of code may be labeled as “pass” if the build or test passes the same code.

4) machinery failure: Machinery failure introduce another potential source of class noise. When a failure in the hardware of CI servers (e.g., a hard drive) or network equipment occurs, the CI pipeline gets disrupted and some running builds or tests may fail the execution. As a result, assigning the execution outcome of builds or tests that failed due to such a disruption would introduce contradictory entries.

5) using test execution outcomes before and after bug fixes: An additional source of class noise arises when different execution results of the same set of tests appear as a result of fixing a bug in a code commit. To illustrate, consider a scenario where a test case, denoted as *tc1*, fails when executed against a particular commit due to an erroneous comparison value used in a conditional statement (e.g., `if(x > -1)`). Now, suppose that a software developer fixes the issue by adjusting the numerical value in the conditional statement to `if(x > 0)`. In this case, both conditional statements, before and after applying the fix, have the same syntactical representation – i.e., both statements contain the same code constructs.

After fixing the issue, *tc1* successfully passes when re-executed. However, if we incorporate multiple instances of the same test case, both before and after bug fixes, for lines of code that have the same syntactical representations such as the case with the `if(x > -1)` and `if(x > 0)` statements, class noise is introduced. Consequently, this situation results in observing the same line of code with both “pass” and “fail” class values.

6) human errors: Human errors introduce another potential source of class noise. Software engineers or testers responsible for assigning class values to lines of code for training ML models may make mistakes or exhibit inconsistencies in their labeling decisions. For example, software engineers might assign different class values to identical lines of code in different commits, even though the code is functionally the same. This inconsistency can introduce class noise in the data.

Sources of attribute noise The definition of attribute noise in this thesis follows the one proposed by Van Hulse et al. [13], which states that attribute noise occurs when one or more attributes in a data entry deviate from the overall distribution of other attributes. The extent of the deviation serves as evidence of noise, with larger deviations indicating a higher likelihood of noise.

In the context of CI, two potential sources of attribute noise were identified:

1) coding style violations: One potential source of attribute noise in CI data can arise when a subset of lines of code diverges from the commonly observed coding style in the majority of similar lines of code. In particular, this type of noise occurs when certain lines of code are inconsistent in formatting, indentation, variable naming conventions, or other stylistic elements that deviate from the established coding standards.

2) measurement instruments: Measurement instruments may produce inaccurate attribute values due to algorithmic flaws. For example, a measurement instrument used to calculate the McCabe complexity may generate incorrect measurements if it fails to correctly identify the correct number of edges in the control flow of the analyzed code, leading to attribute noise.

The research studies presented in this thesis focus on handling the impact of class noise by addressing both contradictory entries and misclassified entries. To handle class noise, a tool was designed to correct the class labels of such entries, and a taxonomy of dependencies was developed to reduce the occurrence of misclassified entries in the data-set. Furthermore, the thesis examines the effect of removing noisy entries that exhibit attribute values that substantially deviate from the majority of entries in the data.

1.1.5 Noise-handling strategies

Existing noise-handling strategies in the machine learning literature can be classified into three broad categories: tolerance, removal, and correction. These categories, as previously introduced in the introduction section, aim at reducing the effect of noise in the training data of ML models and improving their predictive accuracy [14], [15], [16], and [17].

In the tolerance category, noisy entries are retained, and machine learning algorithms are designed to tolerate a certain threshold of noise. Robust algorithms, often utilizing techniques like tree pruning and rule truncation [14], are employed to minimize the negative impact of noise. For example, the C4.5 algorithm prunes statistically insignificant parts of the decision tree to improve model construction [13]. The advantage of tolerance-based approaches is that they eliminate the need for data cleaning, saving time and effort. However, these approaches may experience reduced performance when the noise level exceeds a certain threshold [18].

The removal-based category focuses on identifying and removing noisy entries from the training data-set. Approaches in this category typically follow an iterative process to detect and remove potentially mislabeled entries. However, it is important to note that approaches within this category have a few drawbacks. Firstly, the iterative nature of the process leads to high computational costs. Additionally, there is a risk of mistakenly removing entries that are not actually noisy, thus potentially impacting the integrity of the data-set [19]. While this category of approaches allows explicit detection of potentially noisy data entries, it allows users to decide whether a noisy entry should be removed or retained (e.g., PANDA [13]).

In the correction category, noisy entries are corrected instead of removed. This ensures that no information loss is encountered as a result of removing entries from the data. However, existing correction approaches, such as [20] and [15], often exhibit high time complexity. Additionally, when correcting class labels of noisy entries, there is a risk of introducing bias towards one of the classes. Furthermore, correction approaches typically operate in supervised machine learning environments, making their utility unsuitable when class labels are unavailable [13].

Table 1.1: Advantages and disadvantages of existing noise handling strategies

	Tolerance	Removal	Correction
Pros	- No time is needed to handle noisy entries. - No information loss.	- Explicit detection of noisy entries.	- No information loss.
Cons	- Reduces the performance of classifiers as the noise ratio increases.	- High computational cost to detect and remove noisy entries. points. - Information loss.	- High computational cost to detect and correct noisy entries. - Introduce bias towards one of the classes. - Applicable in supervised classification tasks only.

Table 1.1 provides a summary of the advantages and disadvantages associated with each strategy of noise-handling approaches. Tolerance-based approaches offer the advantage of not requiring additional time for data cleaning and preserving information. However, they experience reduced classifier performance as the noise ratio increases. Removal-based approaches explicitly detect noisy data points but incur computational costs and may lead to information loss. Correction-based approaches preserve information but are computationally expensive, risk introducing bias, and are limited to supervised classification tasks. Understanding these different categories of noise-handling techniques is crucial for software engineers to select the most suitable strategy based on their specific requirements and constraints.

This thesis examines the effectiveness of two removal-based techniques and a correction-based class noise-handling technique. In what follows, we first describe the two removal-based techniques, namely Consensus Filter (CF) and Majority Filter (MF), which are widely used and reported in the literature [21], [22], and [23], then we describe the correction-based technique, namely domain-knowledge-based (DB).

Removal-based techniques: The removal-based techniques examined in this thesis utilize an ensemble of machine learning models, including a univariate decision tree (C4.5), K-Nearest Neighbors (KNN), and linear regression (LR), to classify noisy entries in the training data through a voting mechanism. The techniques employ k-fold cross-validation, where k-1 folds are used for training each model in the ensemble, and the remaining fold is used to label each entry as noisy or clean. After k repetitions, each entry in the entire data-set is assigned a label indicating its noisiness. The decision on which entries to remove is determined through a voting mechanism, with CF being more aggressive and removing a higher proportion of entries compared to MF [24]. Based on the majority voting mechanism presented in [21], an entry is considered noisy if it is tagged as such by more than 50% of the models. Conversely, the consensus filter adopts a more conservative approach, removing entries that are tagged as noisy by one or more models from the data-set.

Correction-based technique: The correction-based technique, also termed the domain-knowledge-based technique, relies on our expertise in the domain of source code changes. Considering the nature of code change data, it is common to observe problematic lines of code in a small fraction of the overall code fragment in code commits. Thus, it is unlikely that every line of code within a commit that is labeled as faulty (negative) requires improvement. Similarly, a line of code that appears in a commit in which all lines are labeled as non-faulty is unlikely to be faulty. Hence, the DB technique ensures to relabel contradictory lines of code from '0' to '1' – if those lines have already been seen as part of positively labeled entries.

The procedure of the technique can be summarized as follows:

1. Each line of code in the original data-set is sequentially assigned a unique 8-digit hash value.
2. An empty dictionary is created to store unfiltered entries.
3. The hashed entries in the original data-set are iterated through, and only syntactically unique entries are saved in the dictionary.
4. For each pair of identical entries, the class values are compared in the original data-set and the dictionary. If the values differ and the class of the entry in the original data-set is annotated as '1', then the class of the corresponding entry in the dictionary is relabeled from '0' to '1'. The entry in the original data-set is then discarded. If both entries have the same class value, the entry from the original data-set is added to the dictionary.

Note that the DB technique can be seen as both removal and corrective to noise, since it 1) removes entries that are identical and not contradictory, and 2) corrects the label of identical entries that first appear in the 'negative' class and then the 'positive class'.

1.2 Related Work

ML-based methods for improving CI processes are shown to be effective at identifying fault-prone code changes in software [25] and [26]. The main advantage that practitioners and researchers seek when using such methods is to feed developers with useful information about the location of faults in software, such that those can be fixed as quickly as possible [27].

In order to leverage these methods, software metrics are used by researchers and practitioners as predictors of fault-prone modules in software. These metrics include object-oriented metrics [28] (e.g., weighted methods per class, number of ancestors of a class), process metrics [29] (e.g., code churns, number of changes made to files in commits), product metrics [30] (e.g., total size of the program, number of lines of code in a commit), and structural metrics [31] (e.g., cyclomatic complexity) offer valuable insights into fault-prone software modules. However, these metrics operate on a module level and do not provide enough semantic information about the code. In other words, we would not know if two programs are equivalent in terms of their fault proneness if they both had a complexity of 1. Thus, additional information is needed to pinpoint exactly which lines of code are faulty. Our work is the first to leverage a software metric (token frequency) that relies on counting the frequency of textual features in software source code changes as predictors of fault-prone code changes.

1.2.1 Machine learning-based approaches in continuous integration

Several researchers have put forth the argument that ML-based methods for fault prediction are considered strong predictors if their Precision, Recall, and Accuracy exceeded 75% [32] and [33]. In the context of CI, this argument seems attainable by several ML-based approaches. For example, Saidani et al. [8] proposed an approach that uses Long Short-Term Memory-based Recurrent Neural Networks model for CI build outcome prediction. The model was trained on sequential data in which each series observation is the history of build results during a specific time period. The time series prediction produced by the model is then used to predict the outcome of future builds. Evaluated on builds records belonging to 10 open source projects, the results showed that the accuracy of the model ranged from 63% to 85%, whereas the F1-score ranged from 22% to 77%.

Chen et al. [34] proposed analyzing build logs and changed files for predicting outcomes of builds in CI. The proposed approach used an adaptive prediction model that switches between two models based on the build outcome of previous builds. The evaluation was performed on 20 projects, and the results have shown that the approach reached 87.4% in Precision, 88.3% in Recall, and 87.4% in F1-score.

Zhang et al. [35] proposed a test selection technique that starts by build prediction. The approach uses 21 software metrics from the TravisTorrent data-set to construct ML models to predict the probability of a specific build failure and transform the probability into test proportion, with respect to a selected test case prioritization technique. Based on the output of the ML model, it selects a prioritized test suite and a variable proportion of test cases with respect to a build. Using 117 projects for the evaluation, the results showed that using the approach improves performance in terms of Recall to 88.9%.

All of these studies evaluated their proposed approaches using information retrieval metrics – such as Precision, Recall, and F1 – and AUC in some cases. However, it is important to account for imbalanced data-sets, since generally, the number of failed builds is less than the passed ones in software projects

[30]. Thus, using an evaluation metric that equally accounts for the failing and passing builds is important to get a better understanding of the model's performance. Our work uses Mathew's Correlation Coefficient to mitigate the risk of reporting inflated results and making over-optimistic conclusions.

1.2.2 Noise-handling in software engineering contexts

There are several noise-handling techniques in the body of literature. The choice of techniques depends on factors such as the nature of the data, the type and ratio of noise, and the specific requirements and domain of the problem being addressed. In this section, we highlight some of the existing removal and correction-based techniques that have been widely used in the literature. Further, we present examples of their application in different software engineering contexts.

Van Hulse and Khoshgoftaar [36] conducted a study to investigate the impact of class noise, particularly when it occurs in the minority class of software quality data. Their findings showed that traditional-based algorithms like Naive Bayes are more effective in handling noise compared to algorithms like Random Forest. In contrast, Folleco et al. [37] reported different results, showing that increasing the level of class noise in the minority class significantly hinders the classification performance of a classifier. Interestingly, their study showed that the most consistent classification performance was achieved using a Random Forest model. These contrasting results regarding the classifier effectiveness highlight the importance of considering the data-set type when determining the most suitable classifier for noise-handling.

Further, Khoshgoftaar and Seliya [38] suggested that focusing on handling noise before training a classifier is more beneficial than focusing on finding the best classifier. They reported that even the best classification algorithm can perform very poorly if the data contained a high level of class noise.

Brodley et al. [15] proposed the Consensus Filter, an ensemble method that employs majority voting to identify and eliminate mislabeled instances. CF utilizes multiple supervised learning algorithms to detect consistently misclassified instances, which are labeled as noisy and removed from the training set. Evaluation results show that when the class noise level is below 40%, employing filtering techniques, such as CF, improves predictive accuracy compared to not filtering the data. This suggests that incorporating any form of filtering strategy is likely to enhance classification accuracy. This approach has been extensively used in the SE literature (e.g., [37] and [39]).

Guan et al. [14] extended the work of Brodley et al. by introducing CFAUD, a variant of CF that incorporates a semi-supervised classification step for predicting unlabeled instances. Their evaluation on benchmark data-sets using three popular machine learning algorithms demonstrates that both majority voting and CFAUD have a positive impact on learning across various noise levels (ranging from 10% to 40%).

Muhlenbach et al. [40] proposed an outlier detection approach that employs neighborhood graphs and cut-edge weight algorithms to identify mislabeled data points. Noisy instances are either removed or relabeled based on the labels of their neighbors. The study shows that using this filtering approach yields better performance in nine out of ten domain data-sets when the noise removal level exceeds 4%.

Khoshgoftaar et al. [41] introduced a rule-based approach for noise detection, using Boolean rules to identify noisy data points. The identified noisy instances are then removed from the data-set before training the model. Comparative results with the CF algorithm by Brodley et al. suggest that the CF algorithm outperforms the rule-based approach in terms of classification accuracy when introducing noise in 1 to 11 attributes at different noise levels.

While the majority of the reported studies provide empirical evidence supporting the handling of both class and attribute noise in data, our research provides counter-evidence related to attribute noise. The findings align with those described in Liebchen et al. [7], which suggest that the definition and impact of noise are highly dependent on the specific domain in which noise occurs. In the context of test case selection, the study suggests that handling attribute noise by identifying outliers in the attributes is not observed to have a detrimental effect.

1.3 Research Focus and Questions

This thesis was organized into several empirical research studies.

The main research question that this thesis addresses is: *How to improve the effectiveness of ML-based methods for continuous integration by handling class and attribute noise?* This research question was motivated by the observation that large amounts of lines in code change data are labeled with inaccurate class values, and similarly contain attribute values that do not accurately characterize the assigned class values. Hence, removing/correcting such inaccurate values can potentially improve the effectiveness of ML-based methods for solving CI tasks.

Therefore, we empirically investigated aspects that concern the effect of noise in CI data and investigated ways to minimize the effect of noise on solving CI tasks. Specifically, our main focus was to understand and improve the prediction performance of ML-based methods in 1) test case selection, 2) build job outcome predictions in continuous integration, and 3) code change request predictions.

To answer the research question, we addressed eight detailed research questions. Figure 1.4 shows these research questions and illustrates how they are structured and related to each other.

The first research question addressed the growing need in the industry to reduce the cost overhead associated with software regression testing. To that end, we developed a tool that analyzes the dependency patterns between historically committed code changes and test case execution results.

Prior to the development of this tool, most existing work in the literature relied on metrics related to the source code (e.g., McCabe complexity), metrics associated with the development processes (e.g., git commits), and metrics derived from test history (e.g., rate of test failures). However, these metrics often lacked sufficient semantic information about the analyzed source code, e.g., we would not know if two programs are equivalent in terms of their fault proneness if they both had a complexity of 1. Therefore, we developed a tool that would allow us to study such kind of dependencies. The tool was founded on the premise that if we could measure the frequency of tokens (e.g., `if`, `for`, `while`) in code commits that had previously triggered test case failures, then we can train a model with such measurements to predict test cases that will react to new code commits. The measurement of token frequency was achieved using a third-party open-source tool, called CCFlex [42], which was shown to provide good results in code analysis tasks. Another goal of designing this tool was to allow us to pinpoint exactly which lines in the code triggered test cases to react. This would allow practitioners to quickly fix faults in their code as soon as they arise.

The following research question was posed:

- *RQ1: How to reduce the number of executed test cases by selecting the most effective minimal test suite when integrating new code churns into the product's main branch?*

This research question provided a basis for understanding that noise in software code change data can adversely affect the predictive performance of ML-based methods. This is because we observed that a lot of identical lines of code in the training data are labeled with different class values. This observation prompted further investigation into whether or not noise has an impact on the predictive performance of the model for test case selection. The results of RQ1 have in turn raised the question of:

- *RQ2: Is there a statistical difference in predictive performance for a test case selection ML model in the presence and absence of class noise?*

We found that there is a statistically significant difference in performance when training a model on data that includes class noise compared to data without class noise. Hence, we explored reasons for introducing class noise in code change and test execution data-set. Our findings showed that the occurrence of class noise can be attributed to the inherent nature of the continuous integration (CI) process – that there are several identical lines in different code commits being integrated. As each line of code in a commit is labeled with the execution result of a test case whose status had changed from pass to fail or vice versa, we encountered a large number of identical lines that were assigned with different class labels.

To address this issue, we developed a class noise-handling algorithm with the goal of reducing the impact of class noise in software code change data on the predictive performance of MeBoTS. Further, we examined the effect of removing instances from the training data that come with high attribute noise values on the performance of MeBoTS. As a result, we posed the following research question:

- *RQ3: How can we improve the predictive performance of a learner for test selection by handling class and attribute noise?*

We found that using an existing attribute noise-handling strategy from the literature [13] for removing instances with attribute noise had no effect on the predictive performance of MeBoTS. As a result, we focused on the issue of class noise. To handle class noise, we implemented a correction-based algorithm that first removes identical lines in the training data and then corrects the label values of contradictory entries. We found that handling class noise using the developed algorithm leads to an improvement in the predictive performance of MeBoTS. This finding has further motivated us to work on reducing the occurrence of class noise in software code change and test execution data.

Since a lot of class noise in the data is introduced due to inaccurate mappings between test execution results and code changes, we wanted to reduce such inaccurate mappings by understanding what types of test cases are sensitive to what types of code changes. By understanding these dependencies, we can map the execution results of sensitive test cases to code changes that appear in code commits and thereby reduce the rate of class noise. Another goal of understanding these dependencies is to assist software testers in determining which types of test cases need to be executed during a CI cycle. Based on these two goals, we posed the following research question

- *RQ4: To which degree do software testers perceive the content of a code commit and test case types as dependent?*

We found that performance-related test cases should be prioritized for execution to test changes related to memory management and algorithmic complexity. These findings are further detailed in Chapter 5.

To validate the relationships identified from the answer to RQ4, we developed a tool that selectively executes test cases that are in relation with code change types that appear in code commits. Then, we measured the total time taken by the tool to perform regression testing and compared it with the time taken by a retest-all approach and the approach employed by our industrial partner. Accordingly, we posed the following research question:

- *RQ5: How to reduce the time of regression testing by selecting only the most relevant test types?*

We found that by using the identified relationships in the answer to RQ4, we could reduce the total regression time compared to both a retest-all approach and the approach employed by our industrial partner, without comprising the effectiveness of testing.

Given these promising results and the positive impact that noise-handling demonstrated in the context of test case selection, we extended our research inquiry and examined the impact of noise-handling on the prediction of several other CI tasks. In particular, we focused on examining noise-handling for an effect on the prediction of build job outcomes, negative code review comments, and code smells. By extending our analysis to these additional CI tasks, we aimed to assess the generalizability and effectiveness of noise-handling approaches across different prediction tasks that can be encountered by software engineers during a CI process.

As mentioned in Section 1.1.1, a CI pipeline typically starts by retrieving the latest code commit submitted to the development repository and then builds the application. The purpose of this build step is to ensure that the code is syntactically correct and that the system has all the required dependencies to function properly. In large and complex projects, performing the build step in CI can take more than 30 minutes to complete [43]. This delay poses a challenge for software engineers, as they have to wait for at least 30 minutes until they know whether their latest code commit will compile successfully. To address this issue and expedite the development process and feature releases, it becomes crucial to minimize the time latency between the CI server and developers without compromising the effectiveness of detecting faults in the code.

Existing research in the literature proposed utilizing process and product metrics to build ML models for predicting whether or not new code changes will compile successfully. However, most of these metrics operate on a file-level, which means that they can only identify the file(s) that are erroneous and would trigger a build failure. In contrast, MeBoTS takes a line-level approach by learning from the frequency of tokens that appear in code commits. This approach allows MeBoTS to pinpoint specific lines of code that triggered a build failure, allowing software engineers to quickly identify and address the problematic lines of code.

Therefore, we aimed at investigating the effectiveness of MeBoTS in predicting the outcome of build outcomes using process, product, and token frequency metrics respectively as predictors of build outcome predictions. Hence, we posed the following research question:

- *RQ6: How effective is the token frequency metric in comparison with traditional software metrics for predicting build outcomes in CI?*

We found that using a line-level metric presents promising potential in improving the prediction of build jobs that will pass, in comparison to 15 other product and process metrics. However, the results also indicated that employing a line-level metric led to a higher rate of false negatives, implying that its effectiveness as a predictor for MeBoTS was inferior to that of file-level metrics. One plausible explanation for this observation could be attributed to the presence of class noise in the data-set.

Since we did not employ any nose handling technique on the data used to answer RQ6, except for the tolerance capability of the ML model in MeBoTS, we needed to further assess how much improvement could be achieved if other noise-handling techniques were applied to the data before training, such as removal and correction strategies. We also needed to examine the impact of the same noise-handling strategies on another type of data-set to reduce the potential for confounding factors related to how the noise was introduced.

To assess how much improvement in the prediction of build outcomes could be achieved by handling class noise, we applied three noise-handling techniques to the training data of build outcomes that we used to answer RQ6 and posed the following research question:

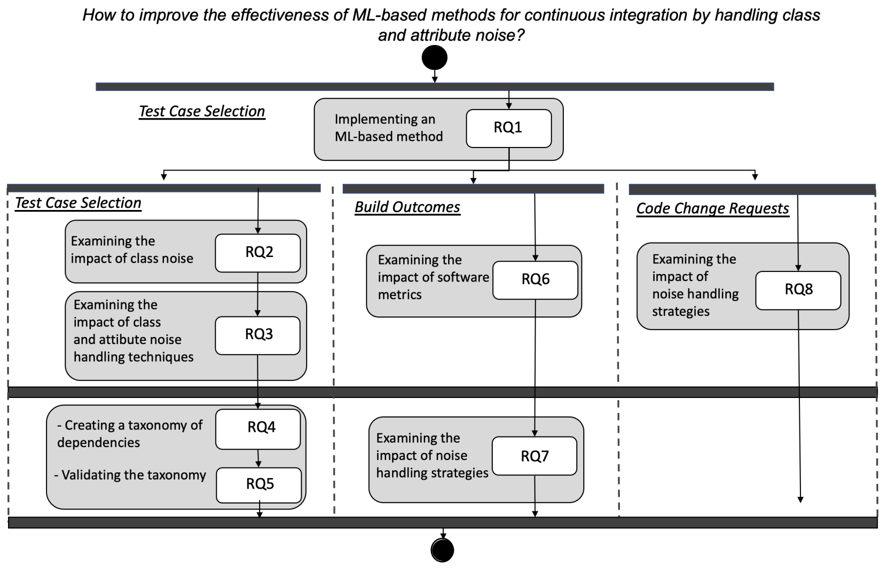

How to improve the effectiveness of ML-based methods for continuous integration by handling class and attribute noise?
Test Case Selection
Implementing an ML-based method
RQ1
Test Case Selection
Examining the impact of class noise
RQ2
Examining the impact of class and attibute noise handling techniques
RQ3
- Creating a taxonomy of dependencies
RQ4
- Validating the taxonomy
RQ5
Build Outcomes
Examining the impact of software metrics
RQ6
Examining the impact of noise handling strategies
RQ7
Code Change Requests
Examining the impact of noise handling strategies
RQ8

Figure 1.4: Mapping between research questions.

- *RQ7: What is the impact of applying class noise handling techniques on predicting the outcome of builds in continuous integration?*

We found that applying the MF and CF techniques to the training data has a statistically significant positive impact on the performance of MeBoTS in predicting build outcomes in CI. These findings are aligned with prior research studies that have investigated the efficacy of these techniques in various software engineering scenarios as well as other domains beyond SE.

To gain a more comprehensive understanding of the effect of noise-handling in CI context, we applied the same three techniques that we used to answer RQ7 to the third type of CI data – code reviews. Therefore, we posed the following research question:

- *RQ8: What is the impact of applying class noise handling techniques on predicting code change requests?*

We found an improvement in the predictive performance of MeBoTS for predicting comments that request a code change after applying the three algorithms to the training data.

1.4 Research Methodology

The research methodology used in this thesis comprises a series of controlled experiments and design science research cycles. Through these two methods, we conducted an in-depth investigation into the effects of noise on CI tasks, explored the application of noise-handling strategies to CI data, and proposed novel approaches to improve the prediction of fault-prone code.

All of the research studies conducted in the course of this thesis can be classified as empirical research, as described in [44]. Table 1.2 provides a mapping between the research questions defined in Section 1.3 and the methodologies used to answer each question.

In this section, we start by summarizing the theory of design science research and controlled experiment respectively. We then describe how we used the two methods in the research studies included in this thesis. Finally, we provide a summary of the contributions that we made in the course of this thesis.

Table 1.2: Mapping between research questions and research methodologies.

Question	Methodology	Paper
RQ1	Design science	A
RQ2	Controlled experiment	B
RQ3	Controlled experiment	C
RQ4	Design science	D
RQ5	Design science	E
RQ6	Controlled experiment	F
RQ7	Controlled experiment	G
RQ8	Controlled experiment	G

1.4.1 Design Science Research

Design science research is the design and investigation of artifacts in a context. Runeson et al. [45] defined design science as an iterative approach consisting of three main activities, namely problem conceptualization, solution design, and empirical validation.

The problem conceptualization is typically the first activity in a DSR. It involves understanding a general problem in terms of a specific problem instance (i.e., in a specific context). During the exploration of the problem instance, it becomes clearer to the researcher and practitioner what the general problem is and consequently what potential solution designs can be made to address the problem. Thus, the solution design activity refers to the mapping between the identified problem instance and the potential solution design. The empirical validation activity concerns assessing whether the designed solution is feasible to solve the identified problem instance.

1.4.2 Controlled Experiments

In software engineering, a controlled experiment is defined as an empirical inquiry that manipulates one variable of the studied setting [46]. Different treatments are applied to different subjects while keeping other variables fixed, and measuring the effects on outcome variables. The purpose is to measure the effect of the treatments on the outcome variables to determine whether there is a causal relationship between them.

In a controlled experiment, the researcher considers the current situation to be the baseline (control), which means that the baseline represents one level of the independent variable, and the new situation that evolves as a result of applying other levels of the independent variable is the one of interest to evaluate. Then the level of the independent variable for the new situation describes how the evaluated situation differs from the control. During these investigations, quantitative data is collected and statistical methods are applied.

Wohlin et al. [46] identified five sequential steps for conducting controlled experiments in SE, as illustrated in Figure 1.5. These steps are described as follows:

1. Scoping: in this step, the objectives of the experiment are defined.
2. Planning: this step concerns identifying the context of the experiment as well as defining the hypothesis, including a null hypothesis and an alternative, the independent and dependent variables, and a suitable design for the experiment.
3. Operation: this step is concerned with the execution and validation of the data. In particular, the focus is to prepare the subjects as well as the tools needed for data collection
4. Analysis and interpretation: this step is concerned with the analysis of the data collected in the operation step. The first step in the analysis is to understand the data by using descriptive statistics. Then we can perform a hypothesis test to determine whether the hypotheses defined in the planning step can be rejected.
5. Presentation and package: this step is concerned with presenting and packaging the findings

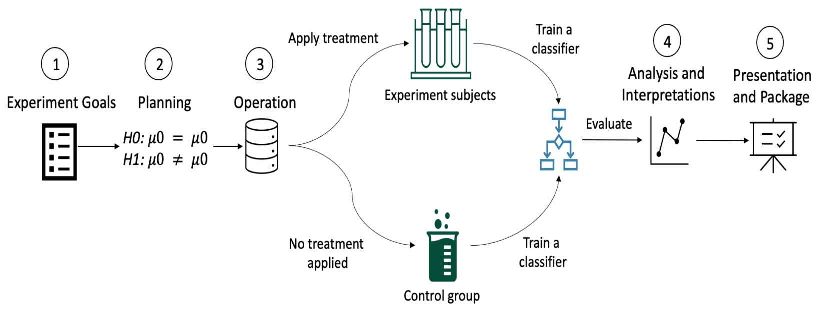


Figure 1.5: An overview of the controlled experiment design used in the thesis.

1.4.3 Research Methods

Throughout the course of this thesis, we conducted a total of seven studies to answer the eight research questions. These studies were conducted using two distinct research methodologies: four studies were designed according to DSR and four studies followed the controlled experiment methodology. The following provides a detailed description of the research method that we followed in each of the eight studies.

1. **Paper A**

   The study presented in Paper A was conducted following the DSR methodology. In order to address the problem of cost overhead in performing software regression testing, several researchers proposed utilizing test prioritization, minimization, and selection techniques [47]. These techniques, however, have inherent limitations that delimit their application in practice. For instance, static analysis-based tools require the code to be compiled in order to access abstract syntax trees and get semantic information about the code. This delimits the applicability of such tools only when the code-base compiles successfully. On the other hand, dynamic analysis tools require real-time test coverage information, which can be demanding and expensive if hardware resources are limited. While measuring code coverage is crucial for determining the extent of code exercised by test cases, it does not guarantee that the system under test is fault-free or fully tested – even with high code coverage, critical defects or untested scenarios may still go unnoticed. These limitations highlight the need for alternative approaches to address the cost overhead in performing software regression testing without compromising the quality of testing.

   To overcome these challenges, we conducted a DSR study (Paper A) at a company that operates in the field of telecommunication and observed their testing workflow. We chose DSR as our methodology because it allowed us to gain practical insights into the problem domain at our industrial partner. Our observation showed that several test cases in the regression suite were detecting faults within the same modules, indicating the presence of redundant tests that are unnecessarily executed. In addition, we observed that the company was executing over 300,000 test case executions on a weekly basis to accommodate their two-week feature-release cycle and that the number of integration tests was increasing rapidly.

   Based on these observations, we designed a solution to the problem posed in RQ1. The solution represents a new ML-based method, called MeBoTS, that operates on a fine-grained level (i.e., line of code level). Unlike static analysis tools, MeBoTS does not require compiling the code base to access abstract syntax trees. Instead, MeBoTS is a language-agnostic [1] solution, which leverages code tokens that appear in code changes as predictors for test case execution results. To assess the effectiveness of MeBoTS in addressing the identified problem, we conducted an evaluation using a data-set from a legacy system developed in-house by the company. The data-set consisted of 82 code revisions and test executions. Two evaluation trials were conducted to assess the performance of MeBoTS.

[1]MeBoTS is language-agnostic, which means that it can operate across multiple programming languages, treating the code as if it were written in natural language

In the first trial, we used a data-set comprising 1.4 million lines of code and 82 test execution results to train and test five ML models in MeBoTS, including three tree-based models and two deep learning networks. The evaluation was based on the Precision and Recall of the predictions made by MeBoTS. In the second trial, ML models were trained and tested exclusively on code check-ins with less than 100,000 lines of code. This trial aimed to evaluate the performance of MeBoTS in scenarios involving smaller code revisions. The evaluation metrics focused on measuring the Precision and Recall of the model's predictions, providing insights into its effectiveness in predicting the impact of code changes on test cases.

2. **Paper B**

   In the analysis phase of the study presented in Paper A, we observed that a large number of identical lines of code were labeled with different class values (i.e., class noise). This led us to design and implement an experiment wherein we could examine the effect of such occurrences of lines in more detail in Paper B.

   The study presented in Paper B was conducted in compliance with the guidelines outlined in Section 1.4.2 for performing controlled experiment research. The objective of the experiment was to examine a causal relationship between class noise in software source code data and the performance of the ML model in MeBoTS. This objective is to understand the ratio in which class noise needs to be handled by testers before training the model in MeBoTS for test case selection.

   In the planning phase, we defined RQ2 and four null hypotheses. These hypotheses were based on the assumption that class noise in software code change data has a detrimental effect on the performance of ML model for test selection. In the operation phase, we utilized a control group with 0% class noise as a baseline for comparison, while six treatment levels of class noise (10%, 20%, 30%, 40%, 50%, and 60%) were seeded into the data. In the analysis and interpretation phase, we tested whether there is a statistically significant difference in the performance of the ML model in MeBoTS when trained on data with and without class noise. We used the Mann-Whitney and Kruskal-Wallis inference tests since the distribution of the evaluation scores were not normally distributed. The results of the tests showed a statistically significant difference in the model's performance when trained on a data-set with 0% class noise and with the six different levels of class noise. We used this finding to formulate our research problem in Paper C.

3. **Paper C**

   The study presented in Paper C was conducted following the DSR methodology outlined in Section 1.4.1 and the controlled experiment guidelines in Section 1.4.2. The study aimed at handling the effect of class noise in software code change data and thereby improving the effectiveness of MeBoTS in test case selection. In addition to handling class noise, we wanted to understand the effect of removing instances in the data with a high attribute noise rate.

In response to these objectives, we began the study by designing a solution to the problem posed in RQ3. The solution was a lightweight tool that relied on our knowledge of source code changes in CI to correct contradictory entries and remove identical lines of code. The solution represents an algorithm that begins by assigning a unique 8-digit hash value to each line of code in the original data-set and creating an empty dictionary to store unfiltered lines of code. Next, the algorithm iterates through the hashed lines in the original data-set and saves syntactically unique lines of code in the dictionary. Finally, the algorithm compares the class labels of each pair of identical lines in the original and dictionary sets. If the class label in the original set is labeled as passed (1) and the same instance in the dictionary is labeled as failed (0), the algorithm relabels the class label of the line in the dictionary from 0 to 1. If both identical lines have a class label of 1, the algorithm skips adding the line from the original set into the dictionary.

In order to assess the effectiveness of the solution, we utilized the code change data that we collected to answer RQ1 and performed the following steps:

- we applied the algorithm to the original data-set, resulting in a data-set of 140,130 lines of code.
- we trained the ML model in MeBoTS using both the original data (before applying the solution) and the class-noise-curated data (after applying the solution).
- we compared the learning performance of the two models in terms of Precision, Recall, and F1.

Similarly, in order to determine whether attribute noise in CI data has an effect on the performance of MeBoTS, we designed and performed a controlled experiment to examine potential causality between attribute noise removal and the predictive performance of MeBoTS.

In the planning phase, we began by reviewing a few related works in the area of attribute noise-handling to explore existing solutions that can be used in our experiment. Among the solutions reviewed, we chose to work with the PANDA algorithm, as described in [13]. We chose PANDA due to its ease of implementation and its suitability for our research objectives. The formulated hypotheses were based on the assumption that removing instances with attribute noise would improve the predictive performance of MeBoTS. In the operation phase, we implemented and applied the PANDA algorithm to the data that we utilized in Paper A. Ten different treatment levels were applied (5%, 10%, 15%, 20%, 25%, 30%, 35%, 40%, 45%, and 50%), each corresponding to a fraction of the instances that were removed before building the ML model in MeBoTS.

In the analysis and interpretation phase, we utilized the Mann-Whitney and Kruskal-Wallis statistical tests to determine whether attribute noise removal has a significant impact on the performance of MeBoTS. The Mann-Whitney test was employed for making pairwise comparisons between the evaluation measures obtained with each treatment level and those at obtained at 0%.

4. **Paper D**
   In the analysis phase of the study presented in Paper C, we observed that utilizing a class noise-handling technique could improve the predictive capabilities of the model in MeBoTS for test case selection. Motivated by this observation, our objective was to further reduce the ratio of class noise by gaining a comprehensive understanding of which test case types are in relation to what specific code changes. This understanding would enable us to build the training data of MeBoTS by accurately mapping code changes with the execution results of tests that are directly related to the respective code change. To that end, we developed a faceted taxonomy that depicts dependency links between code changes and test case types.

   We began the taxonomy building by reviewing a few related works that empirically or theoretically examine the relationship between code change constructs and test case types. Thereafter, we used the outcome of our literature search to seek the opinions of software engineers and testers at four of our collaborating partners about the dependency between types of code changes and test cases. The resulting taxonomy comprised a total of six types of code changes and eight test case types.

   We validated the taxonomy by demonstrating and discussing the orthogonality between strongly dependent test case types and code change types, based on the input given by software engineers and testers. The discussion resulted in a consensus among the recruited participants about the dependencies between two types of code changes and eight test case types. Hence, we validated the dependencies between two types of code changes and their dependent test case types.

5. **Paper E**
   The study presented in Paper E was conducted following the DSR methodology. The study aimed at validating the taxonomy presented in Paper D using the utility demonstration method, as described in [48]. Therefore, we formulated our study problem in RQ5.

   The solution represents a tool that utilizes ML to classify code changes into one of the code change categories illustrated in the taxonomy of dependencies in Paper D. Based on the frequency of occurrence of each type of code change in a new code commit, the tool selectively executes tests that belong to types that are dependent on these changes. This way of selecting test cases eliminates the need of historical data on test case verdicts, which allows software engineers to use the tool from the outset of the software development process.

   The validation of the method was done by measuring the total regression testing time reduced by the tool and its effectiveness in selecting relevant test cases that require executions. The total reduced time was then compared against the time required by a retest-all approach and the approach used at the case company. The validation was done using nine code revisions and 26,576 executions of 868 test cases.

6. **Paper F**
   The study presented in Paper F was conducted following the guidelines of controlled experiments research presented in Section 1.4.2.

   The scope of the experiment was twofold. Firstly, we wanted to investigate the effectiveness of the token frequency metric utilized by MeBoTS as a predictor for build outcome in CI. The goal was to assess whether the token frequency metric can reliably assist software engineers to pinpoint and quickly fix issues in lines of code that trigger build failures. Secondly, we sought to evaluate the effectiveness of the tolerance capability of the random forest model used in MeBoTS for handling noise in the training data. This analysis aimed to determine how well the model could tolerate and mitigate the negative effects of noise on the performance of the model for build outcome predictions.

   In the planning phase, we hypothesized that training a model on the token frequency metric is more effective than file-level metrics for build outcome prediction. In the operation phase, we collected data on build outcomes from a total of 117 Java open-source projects available in the TravisTorrent database [49]. The collected data also comprised fourteen software product and process metrics that we utilized in the analysis to answer RQ6. In the analysis and interpretation phase, we employed the Kruskal-Wallis test to compare the Precision, Recall, F1, and MCC obtained using the process and product metrics and the token frequency metrics. To supplement the analysis, we calculated the effect size between the Precision, Recall, F1, and MCC scores achieved when using the line-level metric and the next most effective file-level software metric.

7. **Paper G**
   The study described in Paper G followed the guidelines for conducting controlled experiments, as presented in Section 1.4.2.

   In Paper C, we only used one technique of class noise-handling in the context of test case selection, which is not sufficient to draw general conclusions about the effectiveness of handling class noise in CI data. Therefore, in this study, the scope was to investigate whether the same results hold for two other noise-handling techniques using other types of CI data. Specifically, the study examined the effects of three different noise-handling techniques on build, code review, and historical code change data. The motivation behind the study was to address the growing demand among software companies to promptly detect and fix faulty code changes while providing constructive feedback to software engineers. Therefore, we formulated our study problems in RQ7 and RQ8.

   In the planning phase, we hypothesized that applying any of the examined class noise-handling techniques to the training data of build outcomes and code review data would improve the predictive performance of MeBoTS, compared to leaving the noise in place and relying solely on the model's ability to tolerate noise.

To address RQ7, we utilized the same data-set of historical build job outcomes that we extracted and introduced in Paper F. The three treatment levels (i.e., noise-handling techniques) were applied to the experiment subjects before being fed as input to MeBoTS for training. The experiment's subjects were generated using 10-fold stratified cross-validation on the control group data.

To study the effectiveness of the selected noise-handling techniques in more contexts, we extended our analysis to another type of software engineering data-set and posed RQ8. There, in the planning phase, we hypothesized that applying the same three noise-handling techniques used in our exploration of RQ7 would have a positive impact on the predictive performance of MeBoTS in code change request predictions.

In the operation phase, we collected historical code review comments and their corresponding code changes from two Java open-source projects submitted to Gerrit, a code review tool. To generate the binary class labels from the collected comments, we manually annotated a sample of the extracted code review comments data from the two collected projects. Subsequently, we trained the model in MeBoTS on the collected code changes and their corresponding review comment labels. We applied 10-fold stratified cross-validation to generate the experiment's subjects and to evaluate the model's performance.

In the analysis and interpretation phase, we tested the hypotheses defined for RQ7 and RQ8 using the Krusal-Wallis test. The goal was to determine whether using any of the class noise-handling techniques had a significant effect on Precision, Recall, F1, and MCC of the model in MeBoTS. We also performed pairwise comparisons to compare the distribution of each dependent variable before and after one treatment level respectively.

1.5 Summary of the Findings

In this section, we provide a comprehensive overview of the main findings and contributions obtained from addressing the eight research questions included in this thesis. These research questions, including a short description of our contributions in each paper, are presented in Table 1.3.

1. **Paper A**

   The first finding from the study presented in Paper A is that training and using ML models in CI context with large commits (over 100,000 LOC) results in low precision (55%) and recall (17.4%). The finding suggests that including large commits in the training data of ML-based methods

Table 1.3: A summary of the contributions of the thesis

No.	**Paper**	**Findings/Contributions**
RQ1	A	• Using revisions of small size for training ML-based methods leads to correctly excluding 80% of tests that will fail. • Using traditional-based ML models exhibits a similar predictive performance as deep-learning models.
RQ2	B	• Encountering a class noise ratio above 20% significantly decreases the predictive performance of MeBoTS for test case selection.
RQ3	C	• Using a domain knowledge-based approach for handling class noise improves the prediction of test cases that require no execution. • Removing instances with attribute noise has no effect on the predictive performance of MeBoTS.
RQ4	D	• Performance-related tests are sensitive to changes related to memory management. • We found that performance-related and maintainability tests are sensitive to changes related to complexity.
RQ5	E	• Selecting test types using the taxonomy of dependencies reduces the total regression testing time. • The dependency links between statement and capacity test cases and memory management code changes need to be refined to incorporate other types of code changes.
RQ6	F	• File-level metrics yield better predictive performance of MeBoTS in build outcome predictions compared to the line-level metric.
RQ7	G	• Applying removal-based techniques for noise-handling improves the predictive performance of MeBoT for build outcomes.
RQ8	G	• Applying the removal-based and correction-based techniques for noise handling improves the predictive performance of MeBoT for code change requests.

increases the probability of encountering noise. The noise, in turn, leads to wrong predictions of test execution results.

The second finding from Paper A is that both traditional tree-based models and deep learning models exhibit similar predictive performance when it comes to predicting test execution results. The Precision scores of the five ML models ranged from 67% to 71%, while the Recall scores ranged from 36% to 49% when trained on code commits containing less than 100,000 lines of code.

2. **Paper B**

   The first finding from the study presented in Paper B is that 80% of the code change data collected during the CI process comes with class noise. The presence of class noise in the data is attributed to the nature of CI, since the labeling of individual lines of code in a commit relies on the execution outcome of a test case from a CI cycle.

   The second finding is that the statistically significant effect of the class noise starts at 20% of the noise. This was evidenced by the observed decreases in Precision by 10%, Recall by 4.5%, F1 score by 10%, and MCC by 16%. These findings indicate the adverse effect of class noise when its ratio exceeds 20% in the data.

3. **Paper C**
   The first finding from the study presented in Paper C is that removing 20% of lines of code that come with the highest attribute noise ratio leads to a 3% decrease in Precision and an 8% decrease in Recall. This finding highlights that testers should not remove lines of code from the data in the interest of cleaning attribute noise.

   The second finding in Paper C is that handling class noise using the domain-knowledge-based tool improves the Precision of MeBoTS from 44% to 81% and its Recall from 17% to 87%. These results suggest that testers can accurately exclude 8 out of 10 passing test cases from the regression suite if they use the domain-knowledge-based tool for handling class noise in code change data. Consequently, testers can reduce the total regression testing time by excluding 70% of test cases that do not reveal faults in code changes (Recall improved from 17% to 87%).

4. **Paper D**
   The first finding from Paper D is that memory management code changes should be tested with performance, capacity, load, stress, soak, or volume test cases. Similarly, we found that complexity code changes should be tested with the same types of test cases as memory management in addition to maintainability tests. Using this dependency information allows testers to reduce the ratio of class noise in code change data by mapping memory management changes in code as well as complexity changes to sensitive types of test cases.

The second finding from Paper D is that there is a lack of consensus among testers regarding the relationship between memory and complexity code changes and security tests. Among the reasons for the lack of consensus were the application domain and the type of programming language used. In particular, 33% of testers who participated in the study perceived security tests to be strongly dependent on memory management changes, since those might lead to memory leaks which in turn might expose the system to security breaches. On the other hand, 50% of participating testers argued that memory leaks result in performance issues rather than security breaches. Further, they linked the sensitivity of security tests to the program domain.

5. **Paper E**
   The first finding from the study presented in Paper E is that using the knowledge derived from the taxonomy of dependencies (Paper D) reduces the total software regression testing time by 52.94% compared to the total time required by the industrial partner's approach. Additionally, when compared to a retest-all approach, we found that using the taxonomy can reduce the total regression time by 15.78%.

   Another finding is that the dependency links between statement and capacity test cases, and memory management code changes need to be refined to incorporate other types of code changes or specific instances of each type. One approach to achieve this refinement is by investigating the relationship between specific instances of memory management and complexity code changes, such as memory leaks, buffer overflows, and so on, and statement and capacity test case types.

   Finally, we found that selecting test types that depend on the two most frequently occurring code change types in a commit results in the highest rate of fault detection. This led to a 22.2% improvement in the rate of fault detection compared to selecting tests that depend on the most frequent code change type.

6. **Paper F**
   The main finding from the study presented in Paper F is that utilizing file-level metrics as predictors for build outcomes is more effective compared to token frequency. This conclusion was supported by the evaluation of the model using different metrics, where the MCC scores were taken into consideration. Specifically, the model trained on token frequency achieved a mean MCC of 0.16, whereas the highest MCC score of 0.68 was obtained when using the file-level metric gh_num_commits_on_files_touched metric. However, it is difficult to establish a causality relationship between the number of commits made on files and build outcomes, as they may both be measuring the same thing. In other words, no commits would lead to no failed builds, and more commits would lead to more failed builds.

   Despite these results, we found that using the token frequency metric leads to higher Precision and Recall compared to when using the file-level metrics. Particularly, the average Precision was at 91% and the average

Recall was at 80%, indicating an improved prediction of passing builds and a reduction of false negatives. The observed discrepancy between F1 and MCC highlights the need to evaluate the predictions of build outcomes in light of the confusion matrix.

7. **Paper G**
   The first finding from the study presented in Paper G is that applying both MF and CF techniques would consistently improve the performance of MeBoTS in predicting build outcomes. In this context, applying MF improves Precision from 90% to 96%, Recall from 76% to 98%, F1 from 82% to 97%, and MCC from 0.13 to 0.58. Similarly, applying CF also showed a significant impact, particularly on Recall (improving from 76% to 96%), F1 (improving from 82% to 94%), and MCC (improving from 0.13 to 0.52).

   The second finding is that applying MF and CF to the training data of code review comments consistently improves the performance of MeBoTS in predicting code change requests. In this context, applying MF was found to improve Precision from 34% to 82%, Recall from 15% to 48%, F1 from 17% to 53%, and MCC from -0.03 to 0.57. Similarly, applying CF also showed to improve Precision from 34% to 70%, Recall from 15% to 56%, F1 from 17% to 60%, and MCC from -0.03 to 0.61.

   The third finding is that using DB would significantly improve the average Recall of MeBoTS (improving from 15% to 25%) for predicting code change requests, but not build outcomes. However, the performance improvement that we gain by applying DB is less than those achieved by applying MF and CF.

   In practical terms, these findings suggest that by applying CF or MF techniques to the training data, MeBoTS can make fewer false predictions about successful and failing builds. Similarly, the prediction accuracy of MeBoTS for predicting code change requests can be improved by exposing the training data to MF, CF, or DB.

1.6 Research Validity

Wohlin et al. [46] identified four types of validity threats to empirical studies in the area of software engineering. In our research, we carefully addressed each of these threats to ensure the validity of our findings.

1.6.1 External Validity

External validity is concerned with generalization. It addresses the question of *is there a relation between the treatment and the outcome that allows the findings to be generalized outside the scope of the current study?*

The external validity of our studies is potentially threatened by the small sample size utilized in the analysis of MeBoTS for test case selection. In Papers A, B, and C, we conducted the analysis on a single industrial project written in the C language, with only twelve test cases, 82 code commits and test executions. Additionally, the study presented in Paper D relied on the opinions of a small number of testers. In the second experiment presented in Paper G, the results were drawn based on the analysis of two open-source projects. Hence, the generalizability of these findings beyond their specific context may be limited due to the small sample size. Therefore, we minimized these threats by randomly selecting the sample of test cases and recruiting testers from various software companies to capture a broader range of perspectives.

1.6.2 Internal Validity

Internal validity concerns aspects in the analysis that indicates a causal relationship between independent and dependent variables, although they are not causal.

The most severe threat to the internal validity of our studies is related to the measurement of token frequency, frequency of code change types, and data collection tools. To minimize the risk of this threat, we carefully inspected the code and tested it using small samples of data points.

Another internal validity is related to the time gap between receiving survey responses and conducting the workshop in the study presented in Paper D. During this interval, participants who responded to the survey and participated in the workshop might have changed their opinions about the dependency patterns. Consequently, there is a possibility of misalignment between the dependency links provided by the participants in the survey and those discussed during the workshop. However, we reduced the likelihood of this threat by thoroughly explaining all code change and test case types that were introduced in the survey and during the workshop.

1.6.3 Construct Validity

Construct validity refers to the degree to which experimental variables accurately measure the concepts they purport to measure.

In our research studies, the main threat to the construct validity is related to whether the execution results of test cases and build outcomes that we use for labeling are due to faulty code changes and not due to flakiness in the selected tests, machinery failures, environment upgrades, etc. We minimized this threat by randomly selecting test execution results from the pool of executed tests at our industrial partners and by analyzing a large sample of build execution outcomes (49,040).

Another threat to the construct validity is related to our measurement of class noise, which is based on calculating the ratio of contradictory entries in the data. Since class noise can occur among entries that are not necessarily contradictory but rather inaccurately labeled, it is possible that our measurement of class noise only captures a fraction of the total ratio of class noise in the data. This implies that if we used alternative metrics for measuring class noise, we might get different measurement results of noise. However, it is not trivial to identify all inaccurately labeled entries in CI data since we do not know the actual cause of noise. Hence, our measurement of noise can be seen as a proxy measure of noise.

1.6.4 Conclusion Validity

Conclusion validity focuses on the ability to draw correct conclusions about the relations between the treatment and the outcome of our study.

One potential threat to the conclusion validity in our research concerns our choice of employing a random forest model for drawing conclusions about the effects of class and attribute noise handling, and software metrics on the predictive performance of MeBoTS in test case selection and build outcome predictions. We minimized this threat in Paper G by training two additional models (XGBoost and a neural network) on the build outcome data. The results showed that random forest outperformed these two models in this context.

Another potential threat to conclusion validity concerns the lack of hyper-parameter tuning performed before training the random forest model. We minimized this threat by conducting additional training trials of the random forest model, where we modified the number of trees from 100 to 300 in the additional trial. The results demonstrated that the predictive performance of the model for test case selection remained similar.

1.7 Discussion

The descriptive statistics and improvement ratios on the predictive performance of ML-based methods for CI are summarized in Table 1.4. All significant findings are marked in bold. The presented summary of the results shows that a significant improvement was achieved by applying the CF and MF algorithms to the build and code change request data (MCC improvement in build predictions: 0.39 for CF vs. 0.45 for MF, MCC improvement in code change request predictions: 0.61 for CF vs. 0.6 for MF). Likewise, an improvement in the predictive performance of MeBoTS was achieved by applying the DB to the test selection data (F1-score improvement: 59%). The hypotheses on the effectiveness of DB on test case selection were not statistically tested since we only worked with a small sample of twelve test cases, which is not a large sample. However, the statistical test results reported in the study presented in Paper B showed that seeding contradictory entries into the data has a negative effect on the performance of MeBoTS in test selection. Hence, removing such entries from the same data-set employed in Paper B with DB would significantly improve the results. Contrary to the findings reported in previous studies [50][51], which concluded that the impact of noise on classifier performance is modest, our findings revealed that the impact of class noise is statistically significant on classification performance when class noise ratio exceeds 20%. In addition, we found that applying noise-handling algorithms to the training data improves classification performance over unhandled noisy data. Sluban et al. [52] showed that applying MF to noisy data leads to high Precision and low Recall. This differs from our findings, which revealed that applying MF improves both Precision and Recall in the context of CI.

The lack of statistical significance in the effectiveness of DB on build outcome and code change request predictions can be possibly attributed to the underlying assumption of DB, which states that faulty labeled entries (labeled with 0) that are identical in features to non-faulty labeled entries (labeled with 1) should have their labels corrected from faulty to non-faulty. This assumption may not hold true at all times and can potentially introduce class noise when faulty entries are incorrectly relabeled to non-faulty. The summary results in Table 1.4 indicates that applying DB to the build and code change data-sets has potentially led to an increase in inaccurate labeling. These inaccurately relabeled entries appear to have negatively influenced the model's ability to learn patterns in the code that trigger build failures and code change requests. This can explain the lack of statistical significance found in Precision, F1-score, and MCC – as summarized in Table 1.4.

Another possible reason for the lack of significance can be due to the balancing algorithm used before applying DB to the data. In Paper C and G, we used the random over-sampling technique [53] to balance the classes in the training data. This technique creates new entries of the minority class in the data to match the number of entries in the majority class. Nevertheless, we cannot assert whether using other over or under sampling techniques will have an impact on the effectiveness of DB, and consequently the performance of the ML model. Therefore, future work needs to investigate the use of different balancing algorithms before applying DB to examine the possible impact on the effectiveness of DB.

As far as attribute noise-handling is concerned, the lack of improvement achieved after applying the PANDA algorithm indicates that removing outliers from code change data is not necessary. In fact, the unchanged predictive performance of MeBoTS (Precision remained at 53%, Recall remained at 88%, and F1-score decreased by 1%) after applying PANDA implies that the tolerance capability of the random forest model can sufficiently handle the effect of outliers without the need to remove them. These results are in line with the conclusions drawn by Brodley and Friedl, and Zhu and Wu [21][54], which suggest that attribute noise is less harmful than class noise on classification performance. It is important to note that in Paper C, we applied the PANDA algorithm to a subset of the data that was exposed to the DB algorithm. Consequently, the initial baseline under the column labeled 'original' differs from the value reported under the 'PANDA' column in Table 1.4.

In terms of the improvement in the regression testing time, the results presented in Paper E demonstrate that executing test cases of types that are in dependency with the most occurring code change types reduces the total time of regression testing compared to a retest-all approach as well as the approach adopted by our industrial partner (regression testing time reduction: from 643,64 to 579.66 hours for retest-all, regression testing time reduction: from 170.01 to 146.72 hours for the approach adopted by our industrial partner). While the study presented in Paper E employs a method that selects test cases of types that are sensitive to the most occurring code changes, it highlights the importance of using the dependency information presented in Paper D for constructing the training data of ML-based methods for test case selection.

Table 1.4: Summary of the effect of noise-handling algorithms on the predictive performance of ML-model for CI.

		Original	**DB**		**CF**		**MF**		**PANDA (Removal ratio: 25%)**	
		value	value	improvement	value	improvement	value	improvement	value	improvement
Paper C (Test Selection)	Precision	44%	81%	37%	-	-	-	-	53%	0%
	Recall	17%	87%	70%	-	-	-	-	88%	0%
	F1-Score	25%	84%	59%	-	-	-	-	66%	-1%
Paper G (Build Outcome)	Precision	90%	89%	-1%	93%	3%	**96%**	**6%**	-	-
	Recall	76%	78%	2%	**96%**	**20%**	**98%**	**22%**	-	-
	F1-Score	82%	82%	0%	**94%**	**12%**	**97%**	**15%**	-	-
	MCC	0.13	0.08	-5%	**0.52**	**0.39**	**0.58**	**0.45**	-	-
Paper G (Change Request)	Precision	34%	39%	5%	**70%**	**36%**	**82%**	**48%**	-	-
	Recall	15%	**25%**	**10%**	**56%**	**41%**	**48%**	**33%**	-	-
	F1-Score	17%	27%	10%	**60%**	**43%**	**53%**	**36%**	-	-
	MCC	-0.03	0.17	20%	**0.61**	**0.64**	**0.57**	**0.6**	-	-

Essentially, when building the training data, historical code changes should be labeled with the execution results of test cases that belong to dependent types. This would potentially reduce the ratio of class noise since it increases the accuracy of labeling.

In general, handling class noise by applying removal-based methods to CI data provides a more reliable improvement in the effectiveness of ML-based methods within the context of CI over DB. Improvements were achieved in the two experiments presented in Paper G after applying MF and CF to the training data and all of the results were statistically significant, except for the Precision after applying CF. This result aligns with earlier findings suggesting that applying CF to noisy data often leads to lower precision improvement compared with recall [22]. Nevertheless, experimental observations also showed that having more diversity in the ensemble of classifiers in CF leads to achieving higher precision [52]. In practical terms, the improvement results – in terms of MCC – imply that software and DevOps engineers who are keen on using ML-based methods for predicting faulty code changes can correctly identify a higher number of actual faults when handling class noise with CF or MF (MCC value for CF in build outcome: 0.52 vs. 0.58 for MF, MCC value for CF in code change request: 0.61 vs. 0.6 for MF). Although an MCC value of 0.6 could still suggest the occurrence of false predictions concerning both faulty and non-faulty code changes, it still implies a substantial improvement in the model's ability to predict the occurrence of faults, compared to -0.03 before noise-handling. Similarly, software engineers can reduce the total regression testing time without compromising the effectiveness of testing by using the taxonomy of dependency for test type selection.

1.8 Conclusion and Future Work

The goal of the research presented in the thesis is to improve the effectiveness of ML-based methods for predicting build outcomes, code change requests, and test case execution outcomes by handling class and attribute noise. To achieve this objective, a series of studies were conducted, as outlined in the previous sections. These studies aimed at examining the effects of noise, developing effective strategies, and evaluating the efficacy of existing approaches in handling noise within the CI context. The culmination of these studies led to the development of innovative methods for deselecting test cases that are unlikely to reveal faults in the code and selecting specific types of test cases that have a higher likelihood of revealing faults. Additionally, a class noise-handling method was devised, leveraging our domain expertise in code changes.

The first method (MeBoTS) relies on measuring the token occurrence within historical code commits. This measurement serves as an input for a machine learning model for identifying patterns in the code that are faulty or non-faulty. Our research demonstrates that the effectiveness of this method can be improved when using small code commits for training. This would contribute to predicting faults in the code and reducing the time required to execute test cases during regression testing. Accordingly, software engineers are encouraged to commit small code changes more frequently during their daily development work to reduce the probability of introducing class noise in code commits.

The second method (HiTTs) is based on measuring the frequency of occurrences of code change types in code commits. This measurement is then used to selectively execute test cases of types that are in dependency with the most occurring code changes. We showed that by using dependency information from the taxonomy presented in Paper D, software engineers can reduce the total time of regression testing without compromising the effectiveness of testing. This, however, requires engineers to continuously and accurately tag their test cases with the correct types during the test creation time. It is important to recognize that several test case types depicted in the taxonomy share common objectives, such as performance, load, soak, stress, volume, and capacity, which involve evaluating the system's performance under different workloads or assessing its ability to handle a large number of requests simultaneously. This can make the task of accurately tagging these overlapping test cases with the correct type challenging. Therefore, software engineers need to familiarize themselves and adhere to the ISO guidelines [55] before tagging test cases.

Finally, we showed that using removal-based techniques for noise-handling improves the performance of MeBoTS in predicting build outcomes and code change requests. While using the domain-knowledge-based method was found to improve the effectiveness of test case selection, our findings revealed that this method does not improve the prediction performance of build outcomes. This disparity in the results between the effectiveness of the tool in test case selection and build outcome predictions can be attributed to several reasons, such as the programming languages, the specific domain of the analyzed applications, and the distinct characteristics of open-source and industrial projects analyzed in our research.

The results presented in this thesis provide opportunities for further research in the context of CI and noise-handling. One direction for future research is to examine the use of different metrics, such as TF-IDF, to extract features from code. This approach goes beyond token frequency and considers the weight assigned to tokens based on their occurrences in various code changes. The results of such studies can then potentially be used to develop an extended version of MeBoTS with other metrics.

Another avenue for future research is to measure the time required by MeBoTS for test case selection and compare it with existing approaches for regression testing. This comparative analysis would shed light on the practical applicability of MeBoTS in CI.

Moreover, an additional avenue for future work is to analyze more dependency relationships between different categories of code changes and test case types. The results of such studies can potentially be used to extend the use of HiTTs to cover a wider range of test case types and code changes. In addition, future work needs to utilize the knowledge derived from the taxonomy of dependencies to train MeBoTS on test cases that belong to one type of test cases. The results of such studies can potentially be used by testers to decide which test cases belonging to one type of test need to be executed or excluded from execution during regression testing.

Regarding noise handling, future empirical studies need to utilize a larger sample of projects for code change request predictions to provide a more comprehensive understanding of the effectiveness of the examined techniques and their generalizability. This would increase our confidence in the applicability of those techniques in the context of CI. Moreover, it is important to examine the effectiveness of additional class and attribute noise-handling techniques beyond those examined in our research. This would help identify the most suitable techniques for specific contexts and improve the overall decision-making of which build and test cases need to be executed. Finally, it would be insightful to examine whether the size of projects and the programming language play a role in influencing the predictive performance of MeBoTS when applying each noise-handling technique. By analyzing these factors, we can gain insights into how project size and programming language influence the effectiveness of noise handling, enabling us to make informed decisions and improve the overall performance of MeBoTS.

Chapter 2

Paper A

Predicting Test Case Verdicts Using Textual Analysis of Committed Code Churns

Al-Sabbagh, K.W., Staron, M., Hebig, R., Meding, W.

In IWSM-Mensura, pp. 138-153. 2019.

Abstract

Background: Continuous Integration is an agile software development practice that produces several clean builds of the software daily. The creation of these builds involves running excessive executions of automated tests, which is hampered by high hardware cost and reduced development velocity.

Goal: The goal of our research is to develop a method that reduces the number of executed test cases at each CI cycle.

Method: We adopt a design research approach with an infrastructure provider company to develop a method that exploits Machine Learning to predict test case verdicts for committed source code. We train five different ML models on two data sets and evaluate their performance using two simple retrieval measures: precision and recall.

Results: While the results from training the ML models on the first data-set of test executions revealed low performance, the curated data-set for training showed an improvement in performance with respect to precision and recall.

Conclusion: Our results indicate that the method is applicable when training the ML model on churns of small sizes.

2.1 Introduction

CI is a modern software development practice, which is based on frequent integration of codes from developers and teams into a product's main branch [56]. One of the cornerstones of its popularity is the promise of higher quality delivered by frequent testing and the ability to quickly pinpoint the code that does not meet quality requirements. To achieve this, CI systems execute tests as part of the integration [57]. However, excessive execution of automated software tests is penalized with high hardware cost and reduced development velocity that may consequently hinder agility and time to market.

In order to address this challenge, a CI system should be able to pinpoint exactly which test cases should be executed in order to maximize the probability of finding defects (i.e. to reduce the "empty" test executions). To achieve this, the CI system needs to be able to predict whether a given test case has a chance of finding a defect or not, or at least whether it will fail or pass – predict the verdict of a test case execution.

We set off to address the problem of predicting test case verdicts by training five ML models on a large data set of historical test cases that were executed against changes made to a software developed at company A. The term "code churn" is defined as a measure that quantifies these changes. Throughout the remaining sections of this paper, we use this term to refer to committed code made during different CI cycles.

Our research is inspired from a previous study conducted by Knauss et el. [58], where the authors explored the relationship between historical code churns and test case executions using a statistical model. Their method used precision and recall metrics in predicting an optimal suite of functional regression tests that would trigger failure. In this paper we expand on that approach by going one step further – conducting a textual analysis of what is the code that is actually being integrated. For example, instead of using code location as the parameter, we use such measures as the number of 'if' statements or whether the code contains data definitions. Similarly, our choice of using code churns is inspired from the work of Nagappan and Ball [59], in which the authors presented a technique for early faults prediction using code churn measures. In their publication, the authors identified a positive correlation between the size of code churns and system defects density. Our method builds on this and uses the Bag of Words (BOW) approach to extract features from code churns. This enables the identification of statistical dependency between keywords and test case verdicts. For example, a churn containing a frequent occurrence of keywords like 'new' or 'delete' might trigger specific tests to fail. More precisely, we aim at investigating the following research question:

> *How to reduce the number of executed test cases by selecting the most effective minimal test suite when integrating new code churns into the product's main branch?*

Our study was conducted in collaboration with a large Swedish-based infrastructure providing company. We study a software product that has evolved over a span of a decade by different cross functional teams. As a result of our study, we present a method that uses ML to predict test case verdicts (MeBoTS).

To address this research question, we conduct a design research study, where we develop a new method and evaluate it on the company's data set. Our method is based on the research by Ochodek et al. [42], which uses textual analyses to characterize source code. Our MeBoTS method builds on that by using historical test verdicts as predicted variables and uses Random Forest algorithms to make the predictions.

The remainder of this paper is organized as follows: Section II provides background information about two categories of ML. The sections that follow provide: an overview of the most related studies in this area, a description of the method that we developed in our study as well as the results, validity analysis, recommendations, and finally, conclusion.

2.2 Background

2.2.1 Categories of Machine Learning

Machine learning is a class of Artificial Intelligence that provides systems the ability to automatically make inferences, given examples relevant to a task [60]. The main advantage of using Machine learning over classical statistical analysis, is its ability to deal with large and complex data-sets [61]. These systems can be classified into four categories depending on the type of supervision involved in training: a) Supervised, b) Unsupervised, c) Semi-supervised, d) Reinforcement Learning [60]. Since we view the problem of predicting test case verdicts as a classification problem, we briefly mention the supervised learning category.

In supervised learning the training data-set fed into the ML model contains the desired solution, called labels. The model tries to find a statistical structure between these examples and their desired solutions [61]. A typical task for this kind of learning is classification.

2.2.2 Tree-based and Deep Learning Models

In Machine learning, a decision tree is an algorithm that belongs to the family of supervised learning algorithms. The algorithm has an inherent tree-like structure and is commonly used for solving classification and regression problems [62]. Starting from the root node, the algorithm uses a binary recursive scheme to repeatedly split each node into two child nodes, where the root node has the complete training sample [63]. The resulting child nodes correspond to features in the training data, whereas the leaf nodes correspond to class labels (binary or multivariate). Other algorithms, such as Random Forest and Adaptive Boosting, use Decision trees as a primary component in their structure. These algorithms build a collection of decision trees, called an ensemble, to increase the overall learning of the classification or regression task at hand [61].
Deep Learning is a branch of ML that was founded on the premise of using

successive learning "layers" to achieve more useful representation of the data [60]. The learning of these successive layers are achieved via models called Neural Networks (NN) [60]. A multilayer NN is one that consists of at least three layers: 1) one input layer, 2) at least one hidden layer, 3) and an output layer of artificial neurons [64]. Similarly, a Convolutional network (CNN) consists of a set of learning layers [60]. The main difference between the two networks is in the way they search for patterns in the input space [61]. More precisely, a CNN works by sweeping a matrix-like window, called filter, over every location in every patch to extract patterns from the input data [61]. As opposed to Decision Trees, ANN have a black box nature, which means that no insight about how their predictions were made can be easily accessible [61]. Nevertheless, the main advantage of using deep learning comes from their ability to handle large and complex data-sets of features.

2.2.3 Code Churns

The amount of changes made to software over time is referred to as code churn [59]. As new churns are added, new risks of introducing defects into the system emerge [65]. According to Y. Shin et al. [65], each check-in made into a version control system includes newly added or deleted code that increase the chances of triggering failures. At some point in time, an evolving system may be vulnerable, on average, to one extra fault for every new additional change [66]. For example, in C programming, the declaration of 'static' local and global variables are among the most confused keywords by developers, as each static local and global declaration has a different effect on how the data will be retained in the program's memory [67].

2.3 Related Work

In the following we discuss related work on the specific use of machine learning for test case selection or prioritization.

2.3.1 ML-based Test-Case Selection

Around 2015/16, we find the first machine learning based approaches for test-case selection. With only 4 studies included in the systematic mapping study by Durelli et al. [68], the use of machine learning for test case prioritization seems to be new.

Busjaeger and Xie [69] present an industrial case study in which a linear model is trained with the SVMmap algorithm using the features Java code coverage, text path similarity, text content similarity, failure history, and test age. The evaluation on the industrial case study, considering 2000 changes and over 45 000 test executions shows an Average Percentage of Faults Detected (APFD) of around 85%.

Chen et al. [70] prioritize test programs for compilers "identifies a set of features of test programs, trains a capability model to predict the probability of a new test program for triggering compiler bugs and a time model to predict the execution time of a test program."

Spieker et al. [71] introduced Retecs, a reinforcement learning-based approach to test case selection and prioritization. Retecs considers duration of a test case's execution, previous last execution and failure history. Online learning is used to improve test case selection between continuous integration cycles. The approach was evaluated on 3 industrial data sets, including together more than 1.2 million verdicts, and achieved a normalized Average Percentage of Faults Detected (APFD) of around 0.4 to 0.8 depending on the data set.

Most recently, Azizi and Do [72] perform test case prioritization by calculating a ranked list of components considering the access frequency of a component as well as a fault risk. The fault risk for each component is thereby predicted using a linear model of change and bug histories. Test cases associated with highly ranked components are prioritized. The approach was evaluated on three web-based systems and where it could reduce the number of test cases by 20% while still finding over 80% of the errors.

Palma et al. [73] replicate and extend a work of Noor and Hemmati [74] and [75], to predict test case failure based on a machine learned model basing on test quality metrics as well as similarity-based metrics.

However, to the best of our knowledge, no other learning-based method works for code-churns. The only exception is one of our previous collaboration with Knauss et al. [58]. The introduced code-churn based test selection method (CCTS) analyzes correlations between test-case failure and source code change. The approach was evaluated in several configurations, leading to results ranging from 26% precision up to a 54% with a 97% recall. We deem these results promising and one of the main motivations for this study.

2.4 Method using Bag of Words for Test Selection (MeBoTS)

The following section is a description of the MeBoTS method used in this research, which comprises of three sequential steps, as shown in Figure 2.1. The method utilizes two Python programs and an open source textual analyzer program, called CCFLEX [42].

2.4.1 Code Churns Extraction (Step 1)

A Python-based code churn extraction program was created to collect and compile code churns committed in the source code repository. The program takes one input parameter: a time ordered list of historical test case execution results extracted from a database, where each element in the list represents an instance of a previously run test case and holds information about: the name

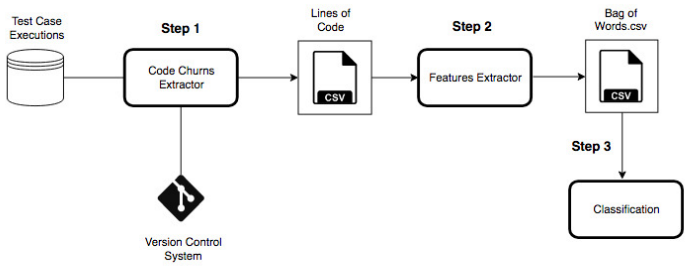


Figure 2.1: The MeBoTS method.

Table 2.1: An Excerpt of the Historical Test Case Executions List

Baseline	Test Case Name	Verdict
ca82a6dff817ec66f	ST-case 22	FAILED
ca82a6dff817ec66f	FT-case 42	PASSED
34bb5e22134200896	FT-case-333	FAILED
34bb5e22134200333	FT-case-3	PASSED

of the executed test case, the baseline code in Git against which the test case was executed, and the verdict value - as shown in Table 2.1.

The program first loops through the extracted list of tests and looks at the change history log maintained by Git and performs a file comparison utility (diff) on a pair of consecutive baselines in the tests list. Note that each baseline value is a hash representation of a revision (build), pointing to a specific location in Git's history log. The result is a fine-grained string that comprises the committed code churns, where each LOC in the churn is compiled with its: 1) filename, 2) physical file path, 3) test case verdict, 4) baseline hash code.

The resulting string is then arranged in a table-like format and written in a csv file, named as 'Lines of Code' in Figure 2.1.

2.4.2 Textual Analysis and Features Extraction (Step 2)

The result of the extract from Git is saved as an array (code churn, filename, physical file path, test case verdict and baseline). This file is the input to our textual analysis and feature extraction. The textual analysis and feature extraction use each line from the code churn and:

- creates a vocabulary for all lines (using the bag of words technique, with a specific cut-off parameter),
- for the words that are used seldom (i.e. fall outside of the frequency defined by the cut-off parameter of the bag of words), a token is created,
- finds a set of predefined keywords in each line,

Table 2.2: Input to the textual analysis and feature extraction

Filename	Path	Content	Hash
firstFile.c	c:/folder	`if (condition == true) printf('Hello World');`	aa00111
firstFile.c	c:/folder	`printf('\n');`	aa00111
secondFile.c	c:/folder	`int i = 10;`	aa00111

Table 2.3: Output from the feature extraction algorithm

Filename	Path	`if`	`int`	`a`	`Aa`	Content
firstFile.c	c:/folder	1	0	3	2	`if(condition == true) printf("Hello World");`
firstFile.c	c:/folder	0	0	2	0	`printf("\n");`
secondFile.c	c:/folder	0	1	1	0	`int i = 10;`

- check each word in the line whether it is part of the vocabulary, it should be tokenized or if it is a predefined feature.

An example input is presented in Table 2.2. The input contains an example code in C.

For the textual analyses, we can pre-define (arbitrarily for this example) two features: "`if`" and "`int`". The bag of words analysis also found the word "`printf`" as frequent. It has also defined the following tokens:

- "`a`" – to denote the words (of any length) that contain only lowercase letters (e.g. "`condition`"),
- "`Aa`" – to denote the words that start with capital letters and continue with lowercase letters (e.g. "`Hello`"),
- "`0`" – to denote the numbers, i.e. sequence of numbers of any length (e.g. "`10`")

The manual features and the bag of words results are then used as features in the feature extraction. Table 2.3, which corresponds to the input from Table 2.2.

Table 2.3 is a large array with the numbers, each representing the number of times a specific feature presents in the line. This way of extracting information about the source code is new in our approach, compared to the most common approaches of analyzing code churns. Compared to the other approaches, MeBoTS recognizes what is written in the code, without understanding of the syntax or semantics of the code. This means that we can analyze each line of code separately, without the need to compile the code or without the need to parse it. This means, that we can take code churns from different files in the same baseline and analyze them together. MeBoTS also goes beyond such approaches like Nagappan et al. [59], which characterizes churns in terms of metrics like number of churned lines or churn size.

Table 2.4: Input to the Classifier Model

File Name	Line Number	F1	F2	F3	F4	F5	..	F500	Verdict
firstFile	1	0	0	6	1	0	..	0	PASSED
firstFile	2	0	0	5	3	2	..	0	PASSED
secondFile	1	0	0	6	1	0	..	0	FAILED

2.4.3 Training and Applying the Classifier Algorithm (Step 3)

We exploit the set of extracted features provided by the textual analyzer in step 2 as the independent variables and the verdict of the executed test cases as the dependant variable, which is a binary representation of the execution result (passed or failed). The MeBoTS method uses a second Python program that utilizes and trains an ML model to classify test case verdicts. The program reads the BOW vector space file in a sequence of chunks, merging the extracted feature vectors and the verdicts vector into a single data frame that gets split into a training and testing set before it is fed into the models for training. Table 2.4 shows an excerpt of the generated data frame.

2.5 Research Design

2.5.1 Collaborating Company

The study has been conducted at an organization, belonging to a large infrastructure provider company. The organization develops a mature software-intensive telecommunication network product. The organization consists of several hundred software developers, organized in several empowered agile teams, spread over a number of continents. Given that they have been early adopters of lean and agile software development methodologies, they have become mature in these areas of work. They have also implemented CI and continuous deliveries.

The organization is also mature with regard to measuring. For instance every agile team, as well as leading functions/roles, uses one or more monitors to display status and progress in various development and devops areas. A well-established and efficient measurement infrastructure, automatically collects and processes data, and then distributes the information needed by the organization.

2.5.2 Dataset

The data-set provided by company A contained historical test case execution results for a mature software product that has evolved for almost a decade. The analyzed product consisted of over 10 thousand test cases and several million lines of code written in the C language. We decided to test the MeBoTS on a set of randomly selected tests that, presumably, reacted to changes in the source code during different CI cycles. Our selection of test cases was based

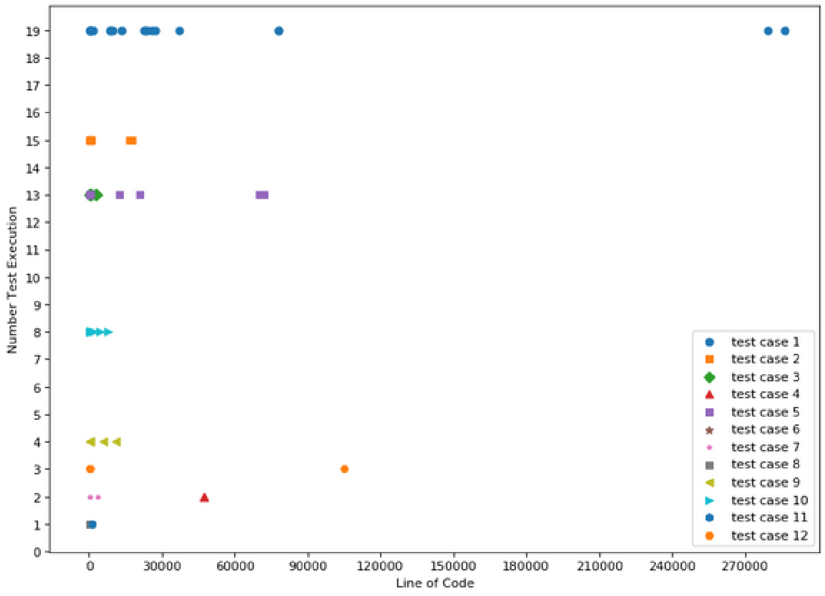


Figure 2.2: Churn Size per Test Execution Plot.

on the granularity of test executions whose verdicts changed from one state to another(see Table 2.1).

The extracted data-set belonged to twelve test cases that were executed 82 times during different CI cycles. The size of the extracted churns was 1.4 million lines of code, among which 618 thousand lines were labeled as passed and 776 thousand as failed. To better understand the shape of the data-set, we visually inspected the size of code churns covered by each test execution. The scatter plot in Figure 2.2 shows the distribution of the 82 extracted test executions, belonging to the twelve test cases. Each mark on the plot represents one test case execution. The x-axis represents the number of lines in the code churn the test case was executed on. The y-axis represents the overall number of test executions for the executed test case. Test executions of the same test case are marked with the same symbol. The visual inspection of the scatter plot suggests that our data-set comprised of churns of varying sizes (large and small). We interpreted this distribution with uncertainty of whether large churns contain additional noise that would adversely affect the training of the ML models. As a result, we decided to curate the original data-set by filtering out tests that were executed on churns whose total size exceeded 110 thousand lines. The visual inspection of the curated data-set is represented in Figure 2.3, which comprised of 290 thousand lines of code, containing a fairly balanced representation of the binary classes (passed and failed), with 110 thousand lines belonging to the passed class and 180 thousand lines belonging to the failed class.

The two data-sets described above were used for training the ML models.

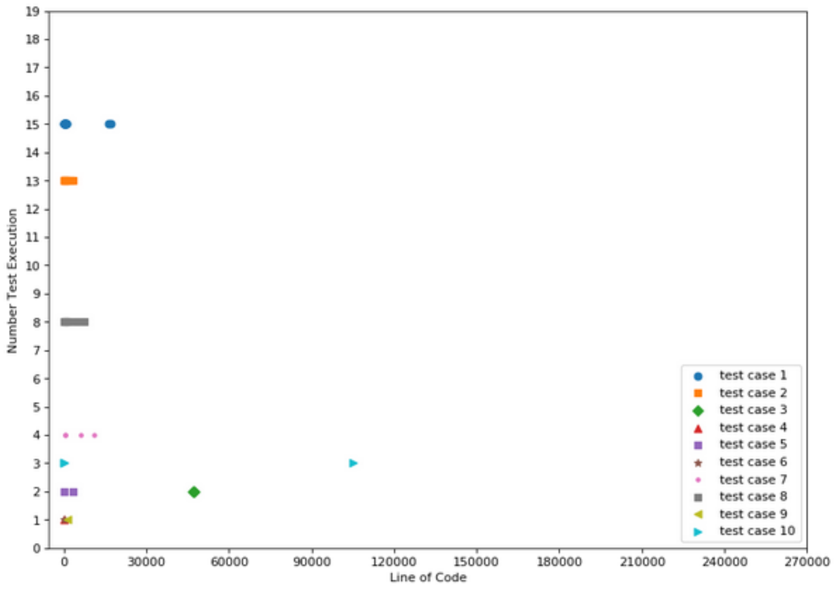


Figure 2.3: Churn Size per Test Execution After Curation Plot.

The first data-set was used in the first phase, whereas the second curated data-set was used in the second phase of ML training, and ultimately became our focus due to data size homogeneity.

2.5.3 Evaluating and Selecting a Classification Model

To select the most suitable model for classifying test verdicts, we selected five different ML models and trained them sequentially. The five models are: 1) Decision Tree, 2) Random Forest, 3) AdaBoost, 4) Multilayer NN, 5) CNN

The choice of selecting the three tree-based models was due to their low computational cost and white box nature, whereas the selection of ANN models were based on their ability to abstract large and complex number of features. Each of the five classification models for test verdicts uses i) the historical test case verdicts ii) the feature vectors in the bag of words table as the baseline of prediction. The evaluation was done in two iterations. In the first iteration, the five models were trained on the original data-set, which contained a mix of large and small churns, comprising a total of 1.4m lines of code for 500 feature vectors. The second iteration involved training the models on the curated data-set, which almost contained 290 thousand lines for the same 500 features, as in the first iteration. Both data sets, curated and original, were split as follow: 70% for the training set and 30% for the test set.

To save the long run time required by automated hyper-parameter tuning tools such as grid and random search, the configuration of the models was done manually. Table 2.5 summarizes the hyper-parameters used for training the

Table 2.5: The Evaluated Models and Their Hyper-Parameters

Classifier	Random State	Number of Trees	Number of Layers	Epochs
DT	123	-	-	-
RF	-	50	-	-
AdaBoost	-	100	-	-
NN	-	-	3	100
CNN	-	-	8	100

models. We used the implementation of Decision Tree, Random Forest, and AdaBoost algorithms available in the Python scikit-learn library [76] and then used the Keras library [77] for the implementation of Multilayer NN and CNN.

The hyper-parameters of the three tree-based models were kept in their default state as found in the scikit-learn library. The only alterations made were in the 'random state' value in Decision Trees and the n_estimator (number of trees) in both Adaboost and Random Forest. With respect to the ANN models, the architecture of the multilayer ANN was represented with three sequential dense layers that consisted of: one input layer, one hidden layer, and one output layer. For the CNN, the stack of layers comprised of: a Reshape layer, a Convolution layer, a Maxpooling layer, and four Dense layers. The learning in both models was induced over 100 epochs (iterations), as can be seen in table 2.5

The performance of the classifiers were evaluated using simple retrieval measure: recall and precision.

These measures are based on the following four categories of errors:

- True positives: correct prediction of test executions that pass
- True negatives: correct prediction of test executions that fail
- False positives: incorrect prediction of test executions that pass
- False negatives: incorrect prediction of test executions that fail

Precision is the number of correctly predicted tests divided by the total number of predicted tests, calculated as follows:

$$precision = \frac{|TruePositive|}{|TruePositive| + |FalsePositive|}$$

Recall is the number of correctly predicted tests divided by the total number of tests that should have been positive.

$$recall = \frac{|TruePositive|}{|TruePositive| + |FalseNegative|}$$

While recall and precision measures relate to one another, precision is a measure of exactness, whereas recall is a measure of completeness indicating the

Table 2.6: Models Evaluation before Data Curation

Model	Precision	Recall	True Neg	False Pos	False Neg	True Pos
DT	43.9%	17.2%	191,883	40,781	153,709	32,003
RF	44%	17.7%	190,864	41,800	152,794	32,918
AdaBoost	51.9%	6.5%	221,472	11,192	173,590	12,122
NN	50.3%	31%	226,994	5,670	179,954	5 758
CNN	50.7%	16.5%	202,745	29,919	154,890	30,822

percentage of all predicted failed tests in our case. Since the goal of this study is to reduce the amount of test executions without altering the effectiveness of testing, we needed to minimize the risk of missing tests that will actually fail and, therefore, accept some false alarms in the prediction of failed tests. Thus, we believe that the measure of the model's precision is more important than its recall.

2.6 Results

To answer the research question, we present the results of using MeBoTS for predicting test case verdicts. We interpret the results of the analysis in light of the reported rates of precision and recall and the values of the four categories of errors: true negatives, false positives, false negatives, and true negatives, as shown in Tables 2.6 and 2.7 list the results of training the five models using the original and curated data sets.

2.6.1 Training the Models on Churns of Varying Sizes

The evaluation of the five models in the first iteration reports a mean precision of 47% and a mean recall of 17%, suggesting that all models achieved low performance. The best result was obtained by the Multilayer NN model with a precision rate of 50.3% and a recall rate of 31%. The precision and recall rates for the five models can be seen in Figure 4. The interpretation of these values suggest that out of the 406,948 lines of code that actually triggered a test case failure, the model correctly predicted test failure for 226,994 lines, whereas it could correctly predict a passing test verdict for 5,758 out of 11,428 lines. Similarly, the results of the four categories of errors for the other models can be seen in Table 2.6 and interpreted in the same fashion.

2.6.2 Training the Models on Churns of Small Sizes

The second iteration of analysis involved training the same set of models on the curated data-set, which excluded tests covering churns with quantities above 110k lines of code. The results, shown in Figure 2.5, indicate an improvement in precision and recall when compared to the results in the first iteration for

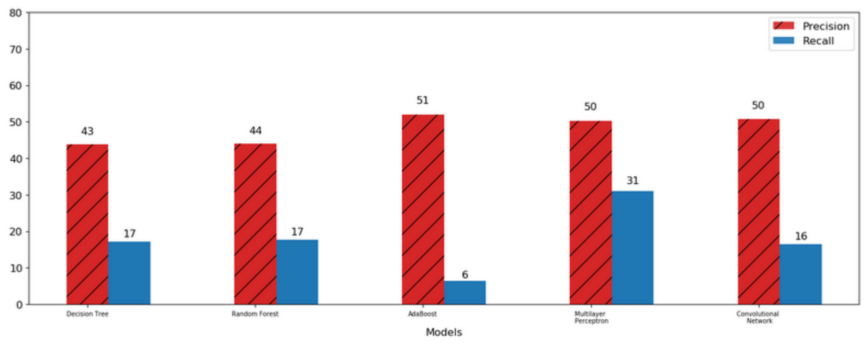


Figure 2.4: Precision and Recall of The Models Before Data Curation

the same types of models. Table 2.7 reports the results from the second round of training, showing a mean precision of 70% and a mean recall of 44.5%. The Multilayer NN model performed best in prediction, such that, it correctly predicted 48,755 lines that actually triggered a test case failure, out of 67,363 lines in the test set.

Table 2.7: Models Evaluation after Data Curation.

Model	Precision	Recall	True Neg	False Pos	False Neg	True Pos
DT	68%	48.4%	46,445	7,548	17,011	15,981
RF	67.9%	49.5%	46,252	7,741	16,637	16,355
AdaBoost	69%	36%	48,656	5,337	21,075	11,917
NN	73.3%	43.6%	48,755	5,238	18,608	14 384
CNN	71.75%	44.9%	48,162	5,831	18,179	14,813

2.6.3 Implication

The results show that we can predict the verdict of a test case with a precision of 73.3%. This means that we can use the results to reduce the test suite by excluding tests that are predicted to pass, but we need to know that there is 26.7% probability that we miss a test case failure. This means that the reduction of the test suite comes with a cost. This cost can be reduced, for example, if we collect the test cases which were not executed and execute them with lower frequency (e.g. during a nightly test suite instead of hourly builds).

2.7 Validity analysis

When analyzing the validity of our study, we used the framework recommended by Wohlin et al. [46]. We discuss the threats to validity in two categories: internal and external. Typically, a number of internal threats to validity emerge in studies that involve designing an ANN architecture, namely that the number of hyper-parameters to tune is large that we cannot cover all combinations

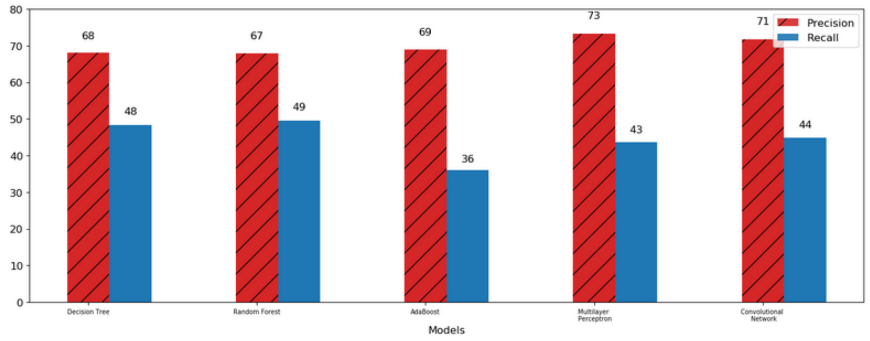


Figure 2.5: Precision and Recall of The Models After Data Curation

to decide on the best configuration for the network. To minimize this threat, we used two different multilayer neural networks and trained them during the two iterations of analysis. This provided us with a sanity check on whether the networks produce similar results. Another threat to internal validity is in the random selection of test cases. There is a chance that the extracted test executions contain one or more test that failed due to factors that do not pertain to functional deficiencies, but due to, for instance, an environment upgrade or machinery failure at execution time. Similarly, there is a chance that the extracted test executions may have failed as a result of defects in the test script code and not the base code. In order to minimize this threat, we collected data for multiple test cases, thus minimizing the probability of identifying test cases which are not representative.

The major threat to external validity for this study comes from the number of extracted tests that were used for training the classifiers. We only studied one company and one product and a limited number of test cases. This was a design choice as we wanted to understand the dynamics of test execution and be able to use statistical methods alongside the machine learning algorithms. However, we are aware that the generalization of the results for different types of systems require further investigations using tests and churns from different systems.

2.8 Recommendations

In this section, we provide our recommendations for practitioners who would like to utilize MeBoTS for early prediction of test case verdicts.

- The choice of using deep learning or tree-based models for solving this supervised ML problem does not lead to better prediction performance. For this reason, we suggest the use of Decision Trees, since they require less computational time and provide knowledge as to how the results of classification were derived.
- We suggest to only utilize code churns that are homogeneous and small in

size prior to applying features extraction with BOW. Small code churns introduce less noise and therefore the quality of the predictions is higher. This can also save practitioners ample time for data curation.

- We recommend that practitioners only extract historical test executions that have failed due to reasons related to functional defects for training the ML model. This knowledge can be obtained from testers/developers who are familiar with the recurrent issues in the source code.

2.9 Conclusion and Future Work

This paper has presented a method (MeBoTS) for achieving early prediction of test case verdicts by training a machine learning model on historical test executions and code churns. We have evaluated the method using two data sets, one containing a variation of large and small churns, and a second containing only small churns. The results from training the models on small churns revealed a precision rate of 73% and a recall of 43.6%, suggesting that the application of the method is promising, yet more investigation is required to validate the findings. Moreover, contrary to other existing methods that use statistical correlations for predicting test verdicts, the main advantage of MeBoTS is the ability to predict verdicts of new code changes as they emerge during development and before they get integrated into the main branch.

We believe that the results of this study open new directions for studies to investigate the effectiveness of MeBoTS on different types of systems using larger set of small churns with more test case executions. Finally, studies that investigate the impact of using different feature extraction techniques, such as word embedding are encouraged to identify any changes in the overall performance of MeBoTS.

Acknowledgment

This research has been partially carried out in the Software Centre, University of Gothenburg, and Ericsson AB.

Chapter 3

Paper B

The Effect of Class Noise on Continuous Test Case Selection: A Controlled Experiment on Industrial Data

Al-Sabbagh, K.W., Staron, M., Hebig, R.

In International Conference on Product-Focused Software Process Improvement. Springer. 2020, pp. 287–303.

Abstract

Continuous integration and testing produce a large amount of data about defects in code revisions, which can be utilized for training a predictive learner to effectively select a subset of test suites. One challenge in using predictive learners lies in the noise that comes in the training data, which often leads to a decrease in classification performances. This study examines the impact of one type of noise, called class noise, on a learner's ability for selecting test cases. Understanding the impact of class noise on the performance of a learner for test case selection would assist testers decide on the appropriateness of different noise handling strategies. For this purpose, we design and implement a controlled experiment using an industrial data-set to measure the impact of class noise at six different levels on the predictive performance of a learner. We measure the learning performance using the Precision, Recall, F1-score, and Mathew Correlation Coefficient (MCC) metrics. The results show a statistically significant relationship between class noise and the learner's performance for test case selection. Particularly, a significant difference between the three performance measures (Precision, F1-score, and MCC) under all the six noise levels and at 0% level was found, whereas a similar relationship between recall and class noise was found at a level above 30%. We conclude that higher class noise ratios lead to missing out more tests in the predicted subset of test suite and increases the rate of false alarms when the class noise ratio exceeds 30%.

3.1 Introduction

In testing large systems, regression testing is performed to ensure that recent changes in a software program do not interfere with the functionality of the unchanged parts. Such type of testing is central for achieving continuous integration (CI), since it advocates for frequent testing and faster release of products to the end users' community. In the context of CI, the number of test cases increases dramatically as commits get integrated and tested several times every hour. A testing system is therefore deployed to reduce the size of suites by selecting a subset of test cases that are relevant to the committed code. Over the recent years, a surge of interest among practitioners has evolved to utilize machine learning (ML) to support continuous test case selection (TCS) and to automate testing activities [78], [79], [80]. Those interests materialized in approaches that use data-sets of historical defects for training ML models to classify source code as defective or not (i.e. in need for testing) or to predict test case verdicts [79], [81], [78].

One challenge in using such learning models for TCS lies in the quality of the training data, which often comes with noise. The ML literature categorized noise into two types: attribute and class noise [82], [83], [18]. Attribute noise refers to corruptions in the feature values of instances in a data-set. Examples include: missing and incomplete feature values [84]. Class noise, on the other hand, occurs as a result of either contradictory examples (the same entry appears more than once and is labeled with a different class value) or misclassification (instances labeled with different classes) [54]. This type of noise is self-evident when, for example, analyzing the impact of code changes on test execution results. It can occur that identical lines are labeled with different test outcomes for the same test. These *identical lines* become noise when fed as input to a learning model.

To deal with the problem of class noise, testers can employ a number of strategies. These can be exemplified by eliminating contradictory entries or re-labeling such entries with one of the binary classes. These strategies have an impact on the performance of a learner and the quality of recommendations of test cases. For example, eliminating contradictory entries results in reducing the amount of training instances, which might lead to a decrease in a learner's ability to capture defective patterns in the feature vectors and therefore decreases the performance of a learner for TCS. Similarly, adopting a relabeling strategy might lead to training a learner that is biased toward one of the classes and therefore either include or exclude more tests from the suite. Excluding more tests in CI implies higher risks that defects remain undetected, whereas including more tests implies higher cost of testing. As a result, it is important for test orchestrators to understand how much noise there is in a training data set and how much impact it has on a learner's performance to choose the right noise handling strategy.

Our research study examines the effect of different levels of class noise on continuous testing. The aim is to provide test orchestrators with actionable insights into choosing the right noise handling strategy for effective TCS. For this purpose, we design and implement a controlled experiment using historical code and test execution results which belong to an industrial software. The specific contributions of this paper are:

- providing a script for creating a free-of-noise data-set which can facilitate the replication of this experiment on different software programs.
- presenting an empirical evaluation of the impact of class noise under different levels on TCS.
- providing a formula for measuring class noise in source code data-sets.

By seeding six variations of class noise levels (independent variable) into the subjects and measuring the learning performance of an ML model (dependent variables), we examine the impact of each level of class noise on the learning performance of a TCS predictor. We address the following research question:

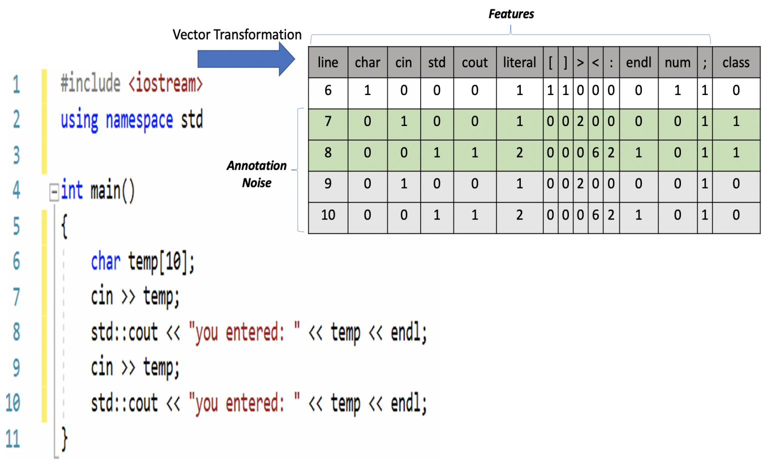
> *RQ: Is there a statistical difference in predictive performance for a test case selection ML model in the presence and absence of class noise?*

3.2 Definition and Example of class Noise in Source Code

In this study, we define noise as the ratio of contradictory entries (mislabelled) found in each class to the total number of points in the data-set at hand. The ratio of noise can be calculated using the formula:

$$\text{Noise ratio} = \frac{\text{Number of Contradictory Entries}}{\text{Total Number of Entries}}$$

Since the contradictory entry can only be among two (or more) entries, the number of all entries for which a duplicate entry exists with a different class label. A duplicate entry is an entry that has the same line vector, but can have different labels. For example, a data-set containing ten duplicate vectors with nine that are labeled `true` and one labeled `false` has ten contradictory entries. It is not trivial to define a general rule to identify which class label is correct based on the number of entries. For example, noise sources might systematically tend to introduce false "false" labels. Since we do not know exactly which class should be used in this context, we cannot simply re-label any instance, as suggested by the currently used solutions (e.g. using majority voting [85] or entropy measurements [52]) and therefore we count all such entries as contradictory. As an illustration of the problem, in the domain of TCS, Figure 3.1 shows how a program is transformed into a line vector and assigned a class label. It illustrates how a data-set is created for a classification task to predict whether lines of a C++ program trigger a test case failure (class 0) or a test case pass (class 1). The class label for each line vector is determined by the outcome of executing a single test case that was run against the committed code fragment in CI. In this study, a class value of '0' annotates a test failure, whereas a class value of '1' annotates a passed test. The Figure shows the



```
1   #include <iostream>
2   using namespace std
3
4   int main()
5   {
6       char temp[10];
7       cin >> temp;
8       std::cout << "you entered: " << temp << endl;
9       cin >> temp;
10      std::cout << "you entered: " << temp << endl;
11  }
```

line	char	cin	std	cout	literal	[	]	>	<	:	endl	num	;	class
6	1	0	0	0	1	1	1	0	0	0	0	1	1	0
7	0	1	0	0	1	0	0	2	0	0	0	0	1	1
8	0	0	1	1	2	0	0	0	6	2	1	0	1	1
9	0	1	0	0	1	0	0	2	0	0	0	0	1	0
10	0	0	1	1	2	0	0	0	6	2	1	0	1	0

Figure 3.1: Class Noise in Code Base.

actual code fragment and its equivalent line vector representation achieved via a statistical count approach (bag-of-words). The line vectors in this example correspond to source code tokens found in the code fragment. Note how lines 5 and 11 are included in the vector representations, since brackets are associated with loop blocks and function declarations, which can be important predictors to capture defective patterns. All shaded vectors in the sparse matrix (lines 7 to 10) are class noise since pairs (7,9) and (8,10) have the same line vectors, but different label class – 1 and 0. The green shaded vectors are 'true labeled instances' whereas the gray shaded vectors are 'false labeled instances'. Note that the Table in Figure 3.1 shows an excerpt of the entries for this example. Since there are 11 lines of code, the total number of entries is 11. The formula for calculating the noise ratio for this example is thus:

$$\text{Noise ratio} = \frac{4}{11} = 0.36$$

If lines 7 to 10 are fed as input into a learning model for training, it is difficult to predict the learner's behavior. It depends on the learner. We also do not know which case is correct – which lines should be re-labelled or whether we should remove these lines. The behavior of the learner, thus, depends on the noise removal strategy, which also impacts the test selection process. If we choose to re-label lines 7 and 8 with class 0 (test case failure), this means that the learner is biased towards suggesting to include the test in the test suite. If we re-label lines 9 and 10 with class 1 (test case pass), then the learner is biased towards predicting that a test case should not be included in a test suite. Finally, if we remove all contradictory entries (7, 8, 9, and 10), then we reduce the learner's ability to capture the patterns in the feature vectors for these lines – we have fewer training cases ($11 - 4 = 7$ cases).

3.3 Related Work

Several studies have been made to identify the effect of class noise on the learning of ML models in several domains [86], [87], [88]. To our knowledge, no study addresses the effect of class noise on the performance of ML models in a software engineering context. Therefore, understanding the impact of class noise in a software engineering context, such as testing, is important to utilize its application and improve its reliability. This section presents studies that highlight the impact of class noise on performances of learners in a variety of domains. It also mentions studies that use text mining and ML for TCS and defect prediction.

3.3.1 The Impact of Noise on Classification Performance

The issue of class noise in large data-sets has gained much attention in the ML community. The most widely reported problem is the negative impact that class noise has on classification performance.

Nettletonet et al. [86] examined the impact of class noise on classification of four types of classifiers: naive Bayes, decision trees, k-Nearest Neighbors, and support vector machines. The mean precision of the four models were compared under two levels of noise: 10% and 50%. The results of the comparison showed a minor impact on precision at 10% noise ratio and a larger impact at 50%. In particular, the precision obtained by the Naive Bayes classifier was 67.59% under 50% noise ratio compared with 17.42% precision for the SVM classifier. Similarly, Zhang and Yang [87] examined the performance of three linear classification methods on text categorization, under 1%, 3%, 5%, 10%, 15%, 20% and 30% class noise ratios. The results showed a dramatic, yet identical, decrease in the classification performances of the three learners after noise ratio exceeded 3%. Specifically the F1-score measures for the three models ranged from 60% to 60% under 5% noise ratio and from 40% to 43% under 30% noise ratio. Pechenizkiy et al. [89] experimented on 8 data-sets the effect of class noise on supervised learning in medical domains. The kNN, Naïve Bayes and C4.5 decision tree learning algorithms were trained on the noisy datasets to evaluate the impact of class noise on accuracy. The classification accuracy for each classifier was compared under eleven class noise levels 0%, 2%, 4%, 6%, 8%, 10%, 12%, 14%, 16%, 18%, and 20%. The results showed that when the level of noise increases, all classifiers trained on noisy training sets suffer from decreasing classification accuracy. Abellan and Masegosa [88] conducted an experiment to compare the performance of Bagging Credal decision trees (BCDT) and Bagging C4.5 in the presence of class noise under 0%,5%,10%,20% and 30% ratios. Both bagging approaches were negatively impacted by class noise, although BCDT was more robust to the presence of noise at a ratio above 20%. The accuracy of BCDT model dropped from 86.9% to 78.7% under a noise level of 30% whereas the Bagging C4.5 accuracy dropped from 87.5% to 77.2% under the same level.

3.3.2 Text Mining for Test Case Selection and Defect Prediction

A multitude of early approaches have used text mining techniques for leveraging early prediction of defects and test verdicts using ML algorithms. However, these studies omit to discuss the effect of class noise on the quality of the learning predictors. In this paper, we highlight the results of some of these work and validate the impact of class noise on the predictive performance of a model for TCS using the method proposed in [78].

A previous work on TCS [78] utilized text mining from source code changes for training various learning classifiers on predicting test case verdicts. The method uses test execution results for labeling code lines in the relevant tested commits. The maximum precision and recall achieved was 73% and 48% using a tree-based ensemble. Hata et al. [79] used text mining and spam filtering algorithms to classify software modules into either fault-prone or non-fault-prone. To identify faulty modules, the authors used bug reports in bug tracking systems. Using the 'id' of each bug in a given report, the authors tracked files that were reported as defective, and consequently performed a 'diff' command on the same files between a fixed revision and a preceding revision. The evaluation of the model on a set of five open source projects reported a maximum precision and recall values of 40% and 80% respectively. Similarly, Mizuno el al. [90] mined text from the ArgoUML and Eclipse BIRT open source systems, and trained spam filtering algorithms for fault-prone detection using an open source spam filtering software. The results reported precision values of 72-75% and recall values of 70-72%. Kim et al. [80] collected source code changes, change metadata, complexity metrics, and log metrics to train an SVM model on predicting defects on file-level software changes. The identification of buggy commits was performed by mining specific keywords in change log messages. The predictor's quality on 12 open source projects reported an average accuracy of 78% and 60% respectively.

3.4 Experiment Design

To answer the research question, we worked with historical test execution data including results and their respective code changes for a system developed using the C language in a large network infrastructure company. This section describes the data-set and the hypotheses to be answered.

3.4.1 Data Collection Method

We worked with 82 test execution results (passed or failed) that belonged to 12 test cases and their respective tested code (overall 246,850 lines of code)[1]. First, we used the formula presented in section 3.2 to measure the level of class

[1]Due to non-disclosure agreements with our industrial partner, our data-set can unfortunately not be made public for replication.

noise in the data-set - this would help us understand the actual level of class noise found in real-world data-sets. Applying the formula indicated a class noise level of 80.5%, with 198,778 points identified as contradictory. For the remainder of this paper, we will use the term ’code changes data-set’ to refer to this data-set. Our first preparation task for this experiment was to convert the code changes data-set into line vectors. In this study, we utilized a bi-gram BoW model provided in an open source measurement tool [42] to carry out the vector transformation. The resulting output was a sparse matrix with a total of 2251 features and 246,850 vectors. To eliminate as many confounding factors as possible, we used the same vector transformation tool and learning model across all experimental trials, and fixed the hyper-parameter configurations in both the vector transformation tool and the learning model (see section 3.5.3)

3.4.2 Independent Variable and Experimental Subjects

In this study, class noise is the only independent variable (treatment) examined for an effect on classification performance. Seven variations of class noise (treatment levels) were selected to support the investigation of the research question. Namely, 0%, 10%, 20%, 30%, 40%, 50%, 60%. To apply the treatment, we used 15-fold stratified cross validation on the control group (see section 3.5.1) to generate fifteen experimental subjects. Each subject is treated as a hold out group for validating a learner which gets trained on the remaining fourteen subjects. A total of 105 trials derived from the 15-folds were conducted. Each fifteen trials was used to evaluate the performances of a learner under one treatment level.

3.4.3 Dependent Variables

The dependent variables are four evaluation measures used for the performance of an ML classifier – Precision, Recall, F1-score, and Matthews Correlation Coefficient (MCC) [91]. The four evaluation measures are defined as follows:

- Precision is the number of correctly predicted tests divided by the total number of predicted tests.

- Recall is the number of correctly predicted tests divided by the total number of tests that should have been positive.

- The F1-score is the harmonic mean of precision and recall.

- The MCC takes the four categories of errors and treats both the true and the predicted classes as two variables. In this context, the metric calculates the correlation coefficient of the actual and predicted test cases for both classes.

3.4.4 Experimental Hypotheses

Four hypotheses are defined according to the goals of this study and tested for statistical significance in section 6. The hypotheses were based on the assumption that data-sets with class noise rate have a significantly negative impact on the classification performance of an ML model for TCS compared to a data-set with no class noise. The hypotheses are as follow:

- *H0p: The mean Precision is the same for a model with and without noise*
- *H0r: The mean Recall is the same for a model with and without noise*
- *H0f: The mean F1-score is the same for a model with and without noise*
- *H0mcc: The mean MCC is the same for a model with and without noise*

For example, the first hypothesis can be interpreted as: *a data-set with a higher rate of class noise will result in significantly lower Precision rate, as indicated by the mean Precision score across the experimental subjects.* After evaluating the hypotheses, we compare the evaluation measures under each treatment level with those at 0% level.

3.4.5 Data Analysis Methods

The experimental data were analyzed using the scikit learn library with Python [76]. To begin, a normality test was carried out using the Shapiro-Wilk test to decide whether to use a parametric or a non-parametric test for analysis. The results showed that the distribution of the four dependent variables did not deviate significantly from a normal distribution (see section 3.6.2 for details). As such, we decided to use two non-parametric tests, namely: Kruskal-Wallis and Mann-Whitney. To evaluate the hypotheses, the Kruskal-Wallis was selected for comparing the median scores between the four evaluation measures under the treatment levels. The Mann–Whitney U test was selected to carry out a pairwise comparison between the evaluation measures under each treatment level and the same measures at a 0% noise level.

3.5 Experiment Operations

This section describes the operations that were carried out during this experiment for creating the control group and seeding class noise.

3.5.1 Creation of The Control Group

To support the investigation of the hypotheses, a control group was needed to establish a baseline for comparing the evaluation measures under the six treatment levels. This control group needs to have a 0% ratio of class noise, i.e. without contradictory entries. To have control over the noise ratio in the treatment groups, these will then be created by seeding noise into copies of the control group data-set (see Section 3.5.2). The classification performance in the treatment groups will then be compared to that in the control group (see Section 3.5.3). In addition, the distribution of data points in the control group is expected to strongly influence the outcome of the experiment. To control for that we aim to create optimal conditions for the algorithm. ML algorithms can most effectively fit decision boundary hyper-planes when the data entries are similar and linearly separable [92]. Therefore, we decided to start from our industrial code changes data-set (See Section 3.4.1) and extract a subset of the data, by detecting similar vectors in the "Bag of Words" sparse matrix. In this study, we decided to identify similarity between vectors based on their relative orientation to each other. What follows is a detailed description of the algorithm used for constructing the control group. The algorithm starts by loading the feature vectors from our industrial code changes data-set and their corresponding label values (passed or failed) into a data frame object. To establish similarity between two vectors we use the cosine similarity function provided in the scikit learn library [76] working with a threshold of 95%. For each of the two classes (passed or failed), one sample feature vector is randomly picked and used as a baseline vector to compare its orientation against the remaining vectors within its class. The selection criterion of the two baseline vectors is that they are not similar. This is important to guarantee that the derived control group has no contradictory entries (noise ratio = 0). Each of the two baseline vectors is then compared with the remaining vectors (non-baseline) for similarity. The only condition for selecting the vectors is based on their similarity ratio. If the baseline and the non-baseline vectors are similar more than the predefined ratio of 95%, then the non-baseline vector is added to a data frame object. Table 3.1 shows the two baseline entries before being converted into line vectors. Due to non-disclosure agreement with our industrial partner, words that are not language specific such as variable and class names are replaced with other random names.

Table 3.1: The Two Baseline Entries Before Coversion

Line of Code	**Class**
measureThreshold(DEFAULT_MEASURE)	1
if (!Session.isAvailable())	0

The script for generating the datasets is found at the link in the footnote[2]. The similarity ratio of 95% was chosen by running the above algorithm a multiple times using five ratios of the predefined similarity ratio. The criterion for selecting the optimal threshold was based on the evaluation measures of a random forest model, trained and tested on the derived control data-set. That is, if the model's Precision and Recall reached 100%, i.e. made neither false positive nor false negative predictions, then we know that control group has reached sufficient similarity for the ML algorithm to work as efficient as possible. The following threshold values of similarity were experimented using

[2]https://github.com/khaledwalidsabbagh/noise_free_set.git

the above algorithm: 75%, 80%, 85%, 90%, and 95%. Experimenting on these ratios with a random forest model showed that a ratio of 95% cosine similarity between the baseline vector and the rest yield a 100% of Precision, Recall, F1-score, and MCC. As a result, we used a ratio of 95% to generate the control group. The resulting group contained 9,330 line vectors with zero contradictory entries between the two classes. The distribution of these entries per class was as follow:

- Entries that have at least one duplicate within the same class: 3,679 entries labeled as failed and 4,280 entries as pass.

- Entries with no duplicates in the data-set: 1 entry labeled as failed and 1,370 entries as passed.

3.5.2 Class Noise Generation

To generate class noise into the experimental subjects, we followed the definition of noise introduced in section 3.2 by carrying out the following two-steps procedure:

1. Given a noise ratio Nr, we randomly pick a portion of Nr from the population of duplicate vectors within each class in the training and validation subjects.

2. We re-label half of the label values of duplicate entries selected in step 1 to the opposite class to generate Nr noise ratio. In situations where the number of duplicate entries in Nr are uneven, we re-label half of the selected Nr portion minus one entry.

In this experiment, a design choice was made to seed each treatment level (10%, 20%, 30%, 40%, 50%, and 60%) into both the training and validation subjects. This is because we wanted to reflect a real-world scenario where the data in both the training and test sets comes with class noise. The above procedure was repeated 15 times for each level, making a total of 90 trials.

A common issue in supervised ML is that the arithmetic classification accuracy becomes biased toward the majority class in the training data-set, which might lead to the extraction of poor conclusions. This effect might be magnified if noise was added without checking the balance of classes after generating noise. In this experiment, due to the large computational cost required to check the distribution of classes across 90 trials, we only checked the distribution under 10% noise ratio. Figure 3.2 shows how the classes in the training and validation subjects were distributed across 15 trials for a 10% noise ratio. The x-axis corresponds to the binary classes and the y-axis represents the number of entries in the training and validation sets. The Figure shows a fairly balanced distribution in the training subjects with an average of 3,421 entries in the passed class and 3,993 entries in the failed class.

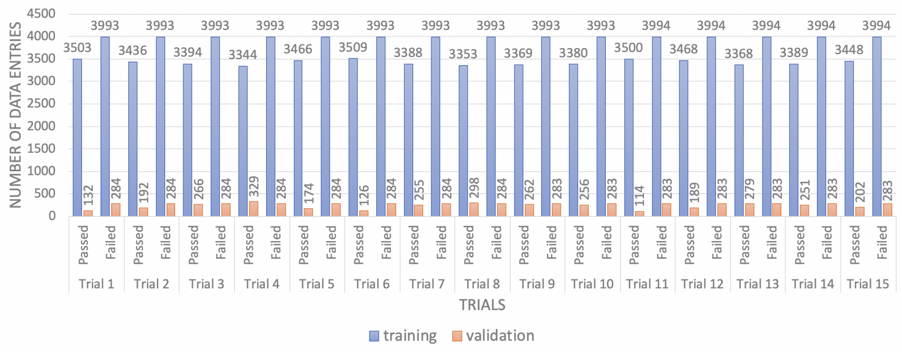


Figure 3.2: The Distribution of The Binary Classes After Generating Noise at 10% Ratio.

3.5.3 Performance Evaluation Using Random Forest

We evaluate the effect of each noise level on learning by training a random forest model. The choice of using a random forest model was due to its low computational cost compared to deep learning models. The hyper-parameters of the model were kept to their default state as found in the scikit-learn library (version 0.20.4). The only configuration was made on the n_estimator parameters (changed from 10 to 100), which corresponds to the number of trees in the forest. We tuned this parameter to minimize chances of over-fitting the model.

3.6 Results

This section discusses the results of the statistical tests conducted to evaluate hypotheses *H0p, H0r, H0f, and H0mcc* and to answer the research question.

3.6.1 Descriptive Statistics

The descriptive statistics are presented in Tables 3.2, 4.5, 3.4, and 3.5 individually for each dependent variable. The values for Precision (Table 3.2), Recall (Table 3.3), F1-score (Table 3.4), and MCC (Table 3.5) are shown for each of the noise ratio (0%, 10%, 20%, 30%, 40%, 50%, and 60%). A first evident observation from the tables is that there is a statistically significant relationship between the mean values of the four dependent variables and the noise ratio, where a lower value of a given dependent variable indicates higher noise ratio. Three general observations can be made by examining the data shown in the four tables:

- There is an inverse trend between noise ratio and learning precision, F1-score, and MCC. That is, when the noise level increases, the classifier trained on noisy instances suffers a small decrease in the four evaluation measures. Figure 3.3 shows this relationship where the x-axis indicates the noise ratio and the y-axis represents the evaluation measures.

- There exists a higher dispersion in the evaluation scores when the noise level increases (i.e. higher standard deviation [SD]).
- The mean difference between the recall values under each noise ratio is relatively smaller than those with the other three dependent variables.

Table 3.2: Descriptive Stats For Precision.

Noise	**N**	**Mean**	**SD**	**SE**	**95% Conf**
0%	15	0.997	0.000	0.000	0.997
10%	15	0.966	0.009	0.002	0.961
20%	15	0.933	0.019	0.005	0.923
30%	15	0.900	0.029	0.007	0.884
40%	15	0.867	0.039	0.010	0.846
50%	15	0.834	0.048	0.012	0.808
60%	15	0.801	0.059	0.015	0.770

Table 3.3: Descriptive Stats For Recall.

Noise	**N**	**Mean**	**SD**	**SE**	**95% Conf.**
0%	15	1.000	0.000	0.000	1.000
10%	15	0.984	0.032	0.008	0.967
20%	15	0.970	0.061	0.015	0.937
30%	15	0.955	0.086	0.022	0.910
40%	15	0.940	0.109	0.028	0.883
50%	15	0.931	0.134	0.034	0.860
60%	15	0.897	0.144	0.037	0.821

Table 3.4: Descriptive Stats For F1-score.

Noise	**N**	**Mean**	**SD**	**SE**	**95% Conf**
0%	15	0.998	0.000	0.000	0.998
10%	15	0.974	0.013	0.003	0.967
20%	15	0.949	0.025	0.006	0.936
30%	15	0.923	0.034	0.008	0.905
40%	15	0.897	0.044	0.011	0.873
50%	15	0.871	0.055	0.014	0.842
60%	15	0.836	0.059	0.015	0.805

3.6.2 Hypotheses Testing

We begin the evaluation of the hypotheses by checking whether the distribution of the dependent variables deviates from a normal distribution. The Shapiro-

Table 3.5: Descriptive Stats For MCC.

Noise	N	Mean	SD	SE	95% Conf.
0%	15	0.996	0.000	0.000	0.996
10%	15	0.946	0.030	0.007	0.930
20%	15	0.894	0.060	0.015	0.863
30%	15	0.841	0.088	0.022	0.795
40%	15	0.790	0.119	0.030	0.727
50%	15	0.742	0.156	0.040	0.660
60%	15	0.674	0.181	0.046	0.579

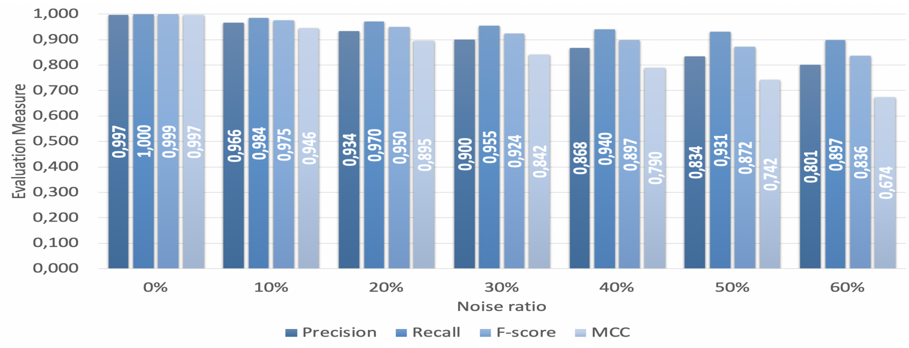


Figure 3.3: Mean Distribution of the Evaluation Measures.

Wilk test results were statistically significant for all the evaluation measures in the majority of the noise ratios. Table 8.12 shows the statistical results of normality for the dependent variables on all noise ratios. These results indicate that the assumption of normality in the majority of the samples can be rejected, as indicated by the p-value (p <0.05) in Table 8.12. Since we have issues with normality in the majority of samples, we decided to run a non-parametric test for comparing the difference between the performance scores under the six noise ratios.

To examine the impact of class noise on the four dependent variables, the Kruskal-Wallis test was conducted. Table 3.7 summarizes the statistical comparison results, indicating a significant difference in Precision, F1-score, and MCC. Specifically, the results of the comparison for precision showed a test statistics of 56.8 and a p-value below 0.001. Likewise, a significant difference in the comparisons between the evaluation measures of F1-score and MCC (F1-score Results: Test Statistics = 54.172, p-value <0.005, MCC Results: Test Statistics = 53.398, p-value <0.005) groups was found. In contrast, no significant difference between the Recall measures was identified.

The Mann–Whitney U test with Precision, F1-score, and MCC as the dependent variables and noise ratio as the independent variable revealed a significant difference (p-value below 0.005) under each of the six levels when compared with the same measures in the no-treatment sample. However, the statistical results for recall only showed a significant difference when the

Table 3.6: Statistical Results For Normality.

	0%	10%	20%	30%	40%	50%	60%
Prec	S=0.59 p<0.005	S=0.82 p=0.02	S=0.87 p=0.11	S=0.91 p=0.28	S=0.91 p=0.32	S=0.88 p=0.13	S=0.92 p=0.40
Recall	S=1.00 p=1.00	S=0.36 p<0.005	S=0.50 p<0.005	S=0.50 p<0.005	S=0.54 p<0.005	S=0.56 p<0.005	S=0.53 p<0.005
F1-score	S=0.59 p<0.005	S=0.78 p=0.009	S=0.67 p<0.005	S=0.74 p=0.003	S=0.83 p=0.037	S=0.69 p=0.001	S=0.8 p=0.02
MCC	S=0.68 p=0.001	S=0.77 p=0.01	S=0.65 p<0.005	S=0.69 p=0.001	S=0.77 p=0.01	S=0.63 p<0.005	S=0.69 p=0.001

Table 3.7: Statistical Comparison Between the Evaluation Measures at All Noise Levels.

	p-value	**statistics**
precision	p<0.005	Statistics=56.858
recall	p=0.164	Statistics=9.180
F1-score	p<0.005	Statistics=54.172
mcc	p<0.005	Statistics=53.398

noise level exceeded 30%. Table 3.8 summarizes the statistical results from the Mann–Whitney test under the six treatment levels. The analysis results from this experiment indicate that *there is a statistical significant difference in predictive performance for a test case selection model in the presence and absence of class noise.* The results from the Kruskal-Wallis test were in line with the expectations for hypotheses *H0p, H0f, H0mcc*, which confirm that *we can reject the null hypotheses for H0p, H0f, H0mcc*, whereas *no similar conclusion can be drawn for hypothesis H0r.* While no significant difference between the recall values was drawn from the Kruskal-Wallis test, the Mann-Whitney test indicates that there is a significant inverse causality between class noise and recall when noise exceeds 30%. In the domain of TCS, the practical implications can be summarized as follow:

- Higher class noise slightly increases the predictor's bias toward the pass class (lower precision rate), and therefore leads to missing out tests that should be included in the test suite.
- A class noise level above 30% has a significant effect on the learner's Recall. Therefore, the rate of false alarms (failed tests) in TCS increases significantly above 30% noise ratio.

3.7 Threats to Validity

When analyzing the validity of our study, we used the framework recommended by Wohlin et al. [46]. We discuss the threats to validity in four categories: external, internal, construct, and conclusion.

Table 3.8: The Comparison Results From Mann-Whitney Test

	10%	20%	30%	40%	50%	60%
Prec	Stat=7.5, p<0.005	S=0.000, p<0.005	S=0.000, p<0.005	S=0.000, p<0.005	S=0.000, p<0.005	S=0.000, p<0.005
Recall	Stat=45, p=0.184	S=40.000, p=0.084	S=40.000, p=0.084	S=35.000, p<0.005	S=30.000, p=0.017	S=25, p=0.007
F1-score	S=7.5, p<0.005	S=0.000, p<0.005	S=0.000, p<0.005	S=0.000, p<0.005	S=0.000, p<0.005	S=0.000, p<0.005
MCC	S=7.5, p<0.005	S=0.000, p<0.005	S=0.000, p<0.005	S=0.000, p<0.005	S=0.000, p<0.005	S=0.000, p<0.005

External Validity: External validity refers to the degree to which the results can be generalized to applied software engineering practice.

Test cases. Since our experimental subjects belong to twelve test cases only, it is difficult to decide whether the sample is representative. However, to increase the likelihood of drawing a representative sample and to control as many confounding factors, we randomly selected a small sample of 12 test cases. Also, the random selection of tests has the potential of increasing the probability of drawing a representative sample.

Control group. The study employed a similarity based mechanism to derive the control group, which resulted in eliminating many entries from the original sample. This might affect the representativeness of the sample. However, our control group contained points that belong to an industrial program, which is arguably more representative than studying points that we construct ourselves. This was a trade-off decision between external and internal validity, since we wanted to study the impact of class noise on TCS in an industrial setting and therefore maximize the external validity.

Nature of test failure. There is a probability of mis-labelling code changes if test failures were due to factors external to defects in the source code (e.g., machinery malfunctions or environment upgrades). To minimize this threat, we collected data for multiple test executions that belong to several test cases, thus minimizing the probability of identifying tests that are not representative.

Internal Validity Internal validity refers to the degree to which conclusions can be drawn about the causal effect of independent on dependent variables.

Instrumentation. A potential internal threat is the presence of undetected defects in the tool used for vector transformation, data-collection, and noise injection. This threat was controlled by carrying out a careful inspection of the scripts and testing them on different subsets of data of varying sizes.

Use of a single ML model. This study employed a random forest model to examine the effect of class noise on classification performances. However, the analysis results might differ when other learning models are used. This was a design choice since we wanted to study the effect of a single treatment and to control as many confounding factors as possible.

Construct Validity Construct validity refers to the degree to which experimental variables accurately measure the concepts they purport to measure.

Noise ratio algorithm. Our noise injection algorithm modifies label values without tracking which entries that are being modified. This might lead to relabeling the same duplicate line multiple times during noise generation. Consequently, the injected noise level might be below the desired level. Thus, our study likely underestimates the effects of noise. However, the results still allowed us to identify a significant statistical difference in the predictive performance of TCS model, thereby to answer the research question.

Majority class problem. Due to the large computational cost required to check the balance of the binary classes under the six treatment levels, we only checked for the class distributions for one noise level - 10%. Hence, there is a chance that the remaining unchecked trials are imbalanced. Nevertheless, the downward trend in the predictive performances as noise ratio increases indicates that the predictor was not biased toward a majority class.

Conclusion Validity Conclusion validity focuses on how sure we can be that the treatment we use really is related to the actual outcome we observe.

Differences among subjects. The descriptive statistics indicated that we have a few outliers in the sample. Therefore, we ran the analysis twice (with and without outliers) to examine if they had any impact on the results. Based on the analysis, we found that dropping the outliers had no effect on the results, thus we decided to keep them in the analysis.

3.8 Conclusion and Future Work

This research study examined the effect of different levels of class noise on the predictive performance of a model for TCS using an industrial dataset. A formula for measuring the level of class noise was provided to assist testers gain actionable insights into the impact of class noise on the quality of recommendations of test cases. Further, quantifying the level of noise in training data enables testers make informed decisions about which noise handling strategy to use to improve continuous TCS if necessary. The results from our research provide empirical evidence for a causal relationship between six levels of class noise and Precision, F-score, and MCC, whereas a similar causality between class noise and recall was found at a noise ratio above 30%. In the domain of the investigated problem, this means that higher class noise yields to an increased bias towards predicting test case passes and therefore including more of those tests in the suite. This is penalized with an increased hardware cost for executing the passing tests. Similarly, as class noise exceeds 30%, the prediction of false alarms with the negative class (failed tests) increases.

There are still several questions that need to be answered before concluding that class noise handling strategies can be used in an industrial setting. A first question is about finding the best method to handle class noise with respect to efficiency and effectiveness. Future research that study the impact of attribute noise on the learning of a classifier and how that compares with the impact of class noise are needed. Other directions for future research include evaluating the level of class noise at which ML can be deemed useful by companies in predicting test case failures, evaluate the relative drop of performance from a random sample of industrial code changes and compare the performance of the learner with the observations drawn from this experiment, study and compare the effect of different code formatting on capturing noisy instances in the data and the performance of a classifier for TCS. Finally, we aim at comparatively exploring the sensitivity of other learning models to class and attribute noise.

Chapter 4

Paper C

Improving test case selection by handling class and attribute noise

Al-Sabbagh, K.W., Staron, M., Hebig, R.

Journal of Systems and Software, 183, 111093.

Abstract

Big data and machine learning models have been increasingly used to support software engineering processes and practices. One example is the use of machine learning models to improve test case selection in continuous integration. However, one of the challenges in building such models is the large volume of noise that comes in data, which impedes their predictive performance. In this paper, we address this issue by studying the effect of two types of noise, called class and attribute, on the predictive performance of a test selection model. For this purpose, we analyze the effect of class noise by using an approach that relies on domain knowledge for relabeling contradictory entries. Thereafter, an existing approach from the literature is used to experimentally study the effect of attribute noise removal on learning. The analysis results show that the best learning is achieved when training a model on class-noise cleaned data only – irrespective of attribute noise. Specifically, the learning performance of the model reported 81% precision, 87% recall, and 84% F1-score compared with 44% precision, 17% recall, and 25% F1-score for a model built on uncleaned data. Finally, no causality relationship between attribute noise removal and the learning of a model for test case selection was drawn.

4.1 Introduction

Machine Learning (ML) models have been increasingly used for automating software engineering activities [80], [93]–[95]. One example for the use of ML models is optimizing software regression testing in continuous integration (CI), where ML is used to recommend which test cases should be included in test suites to reduce the cost overhead for testing resources. Since regression testing is performed frequently (after every commit), they result in large quantities of data that include test execution results. This poses an opportunity to utilize ML when such large data is available for analyses.

Figure 4.1 illustrates a CI pipeline that includes a number of accrued test suites of different sizes - the every-build, daily, and weekend. These suites are organized to regressively verify that no new faults in the system arise as a consequence of new code check-ins, with the goal of reducing the cost of regression testing. The CI system tries to identify and select a small subset of test cases from the pool of available tests to perform regression testing. These test cases are added to the every-build suite and get executed as soon as new code check-ins are submitted to the code repository. Failure to detect faults in this early phase of testing will prolong their discovery until larger suites (the daily or weekend suites) are executed.

Orchestrating test cases in this way allows for an increased development speed and a reduced cost of regression testing, since faults are being continuously discovered and fixed as soon as they are introduced into the code base. Figure 4.1 exemplifies a scenario where the CI system misses adding test case 2 (*Tc2*) to the build suite. This gets penalized by an increased time of testing and faults fixing, since (*Tc2*) will get executed in the daily or the weekend suite. Therefore, it is important to find an effective approach for test case selection to maximize the probability of detecting faults as early as new code check-ins are made.

Several approaches in the literature sought to address the problem of defects prediction and test case selection in CI. Examples include static code analysis [96], [97], static code metrics [74], [59], natural language processing (NLP) [78], [98]. These approaches use data-sets with historical defects for training machine learning (ML) models to classify code as either non-defective or defective (i.e. in need for testing) or to predict whether test cases will fail. In our previous work [78], we studied an industrial case of the use of ML classifiers and textual analysis to predict test case execution results. The method was evaluated on a data-set whose size was 1.4m lines of code (LOC).

However, one of the challenges in building a learner for predicting test case execution results lies in the amount of noise that comes in the data. This challenge is particularly important in the domain of testing, since frequent automated executions of test cases can introduce noise in an uncontrolled way. A complete taxonomy of noise types is still an open research issue [41]. However, two categories of noise types are most commonly addressed in the literature - class and attribute noise [14], [40], [54], [99]. Class noise (also known as label or annotation noise) occurs as a result of either contradictory entries or mislabeling training entries [54], whereas attribute noise occurs due

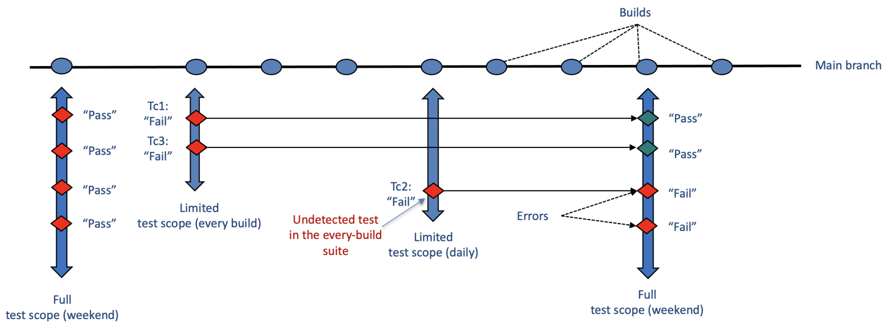


Figure 4.1: CI Pipeline with Test Case Selection.

to either selecting attributes that are irrelevant for characterizing the training instances and their relationship with the target class, or using redundant or empty attribute values [54], [100].

In the domain of TCS, the class noise can be observed when, for example, a code line in the data appears more than once with different class labels (test outcomes) for the same test. These *duplicate appearances* for the same line become class noise for predictors and would consequently hamper their classification accuracy. Similarly, one example of attribute noise in the same domain (TCS) occurs when similar code lines are written in different coding styles. Code lines written in the less frequently used style will be characterized with attributes whose frequency deviates from similar code lines written in the majority styles. These deviations can make code lines written in the less frequently used coding styles become outliers in the data at hand and thereby can negatively affect the learning performance.

A number of research studies proposed several techniques for handling class and attribute noise [14], [15], [16], [17]. Those can be classified into three broad categories: tolerance, elimination/filtering, and correction/polishing. In the tolerance category, imperfections in the data are dealt with by leaving the noise in place and designing ML algorithms that can tolerate a certain amount of noise. Approaches in the elimination category seek to identify noisy entries and remove them from the data set. Entries that are suspected to be spurious (e.g., mislabelled or redundant) are discarded and removed from the training data. In the last category, instead of removing the corrupted entries, those entries get repaired by replacing their values with more appropriate ones. There are a number of advantages and disadvantages associated with each one of these approaches. In the tolerance category, no time needs to be invested on cleaning the data, but a learner built from uncleaned data might be less effective. By filtering noisy instances, we compromise information loss in the interest of retaining cleaner instances of the data. By carrying out correction of noisy instances, we introduce risks of presenting undesirable attributes but preserve maximal information in the data.

In a previous work [98], we introduced an approach for addressing the problem of annotation noise by relabeling contradictory entries and removing duplicate ones in one of the classes. The empirical evaluation of applying the technique was measured with respect to precision, recall, and F1-score using an industrial data. In this study, we extend that work by examining the effect of applying an attribute noise elimination approach, called Pairwise Attribute Noise Detection Algorithm (PANDA) [13], on the performance of a TCS learner using the class-noise cleaned data reported in [98]. The purpose is to provide testers with insights into choosing the right noise handling strategies and to counteract exhaustive efforts on noise cleaning for more effective TCS. For this, we design and implement a controlled experiment on the same industrial data-set used in our previous study [98] and examine the effect of removing training observations that come with high attribute noise on learning. Specifically, we address the following research question:

RQ: How can we improve the predictive performance of a learner for test selection by handling class and attribute noise?

In this study, we focus on examining the effect of handling both class and attribute noise on the performance of an ML classifier for selecting regression tests on both functional and integration testing levels. The sample data-set used belongs to a large telecommunication program written in the C language and consists of 82 test execution results for twelve test cases. We validate the findings by comparing the performance results of three learners with respect to precision, recall, and F1-score. These learners are trained on: original (uncleaned data), class-noise cleaned data, and class and attribute noise cleaned data.

Hereafter, Section 2 will correspond to the related work highlighting studies made on class and attribute noise handling. Then, Section 3 presents background information, providing core concept, a description of the TCS method used in the paper, and examples and definitions on class and attribute noise in code changes data. Section 4 describes the two approaches used in this study for handling class and attribute noise. Section 5 describes the research methodology. Then, Section 6 presents the evaluation results of the effect of class and attribute noise. Thereafter, Section 7 answers the research question and presents recommendations to testers. Section 8 addresses the threats to validity of this study. Finally, Section 9 concludes the findings and highlights future work.

4.2 Related Work

Many research efforts have been made to handle class and attribute noise for improving the predictive quality of learners. However, studies that investigate the impact of class and attribute noise handling in the domain of software engineering is lacking [7]. In this section, we begin by highlighting work that leverage the use of ML models for early prediction of defects and test case verdicts for test selection. Thereafter, we highlight related work that examine the effect of class and attribute noise on learning performances.

4.2.1 Text Mining For Defect Prediction and Test Case Selection

Table 4.1: Results Summary For Defects Prediction and Test case Selection Research

Study	Type	Systems	Predictors	Results
[79]	Defects Prediction	BIRT, ECLP, MODE, TPTP, and WTP	Spam Filter	Precision 40%, Recall 80%
[90]	Defects Prediction	ArgoUML and BIRT	Spam Filter	Precision 72%, Recall 70% Precision 75%, Recall 72%
[81]	Defects Prediction	JHotDraw and DNS	Regression, ADABoosting, C4.5, SVM, K-NN	K-NN: Precision 59%, Recall 69% Precision 59%, Recall 23%
[80]	Defects Prediction	Apache 1.3, Bugzilla, Columba, Gaim, GForge, JEdit, Mozilla, Eclipse JDT, Plone, PostgreSQL, Scarab, and Subversion	SVM	Accuracy 78%, Recall 60%
[78]	TCS	Industrial Software	RF	Precision 73%, Recall 48%

A multitude of early approaches have used text mining techniques for leveraging early prediction of defects and test case verdicts using various learning algorithms and statistical approaches. However, these studies omit to discuss the effect of class noise on the quality of the learning predictors. As a result, in this paper, we mention some of these previous work, as summarized in Table 4.1

A previous work on test case selection [78] utilized text mining from source code changes for training various learning classifiers on predicting test case verdicts. The method uses test execution results for labelling code lines in the relevant tested commits. The maximum precision and recall achieved was 73 and 48 percent using a tree-based ensemble. Hata et al. [79] used text mining and spam filtering algorithms to classify software modules into either fault-prone or non-fault-prone. To identify faulty modules, the authors used bug reports in bug tracking systems. Using the 'id' of each bug in a given report, the authors tracked files that were reported as defective, and consequently performed a 'diff' command on the same files between a fixed revision and a preceding revision. The evaluation of the model on a set of five open source projects reported a maximum precision and recall values of 40 and 80 percent respectively.

Similarly, in an earlier work, Mizuno el al. [90] mined text from the ArgoUML and Eclipse BIRT open source systems, and trained spam filtering algorithms for fault-prone detection using an open source spam filtering software. The results reported a precision of 72-75 percent and a recall of 70-72.

Aversano et al. [81] extracted a sequence of source code snapshots from two version control systems and trained five learning algorithm to predict whether new code changes are defective or not. The K-Nearest Neighbor predictor performed better than the other algorithms with a good trade-off between precision and recall, yielding precision and recall values of 59-69 percent and 59-23 percent respectively.

Kim et al. [80] collected source code changes, change metadata, complexity metrics, and log metrics to train an SVM model on predicting defects on file-level software changes. The identification of buggy commits was performed by mining specific keywords such as 'Fixed' or 'Bug' in change log messages. Once a keyword is found, the assumption that changes in the associated commit comprise a bug fix is made, and hence used for labelling code instances in the relevant commit. The predictor's quality on 12 open source projects reported an average accuracy of 78 and 60 percents respectively.

4.2.2 Class Noise Handling Research

Brodley et al. [15] uses an ensemble of classifiers, named Consensus Filter (CF), to identify and remove mislabeled instances. Using a majority voting mechanism with the support of several supervised learning algorithms, noisy instances are identified and removed from the training set. If the majority of the learning algorithms fail to correctly classify an instance, a tag is given to label the misclassified instance as noisy and later tossed out from analysis. The evaluation results show that when the class noise level is below 40%, filtering results in better predictive accuracy than not filtering. On the basis of their experiments, the authors suggest that using any types of filtering strategies would improve the classification accuracy more than not filtering.

Al-Sabbagh et al. [101] conducted a controlled experiment to examine the effect of class noise at six levels on the learning performance for a test selection model. The analysis was done on an industrial data for a software program written in the C++ language. The results revealed a statistically significant relationship between class noise and the precision, F1-score, and Mathew Correlation Coefficient under all the six noise levels. Conversely, no similar relationship was found between recall and class noise under 30% noise level. The conclusion drawn suggested that higher class noise ratio leads to missing out more tests in the predicted subset of test suite. Moreover, it increases the rate of false alarms when the class noise ratio exceeds 30%.

Guan et al. [14] introduced CFAUD, a variant of the approach proposed by Brodley et al. [15], which involves a semi-supervised classification step in the original approach to predict unlabeled instances. The approach was tested for an effect on learning for three ML algorithms (1-NN, Naive Bayes, and Decision Tree) using benchmark data-sets. The empirical results indicate that both majority voting and CFAUD have a positive effect on the learning of the three ML algorithms under four noise levels (10%, 20%, 30%, and 40%). However, averaged on the four noise levels, the improvement that CFAUD provides over CF is around 12% for each of the three classifiers.

Muhlenbach et al. [40] introduced an outlier detection approach that uses neighbourhood graphs and cut edge weight algorithms to identify mislabeled entries. Instances identified as noisy are either removed or relabeled to the correct class value. Relabeling is done for instances whom neighbours are correctly labeled, whereas entries whom neighbouring classes are heterogeneously distributed get eliminated. Evaluated on ten domains from a machine learning repository, three 1-NN models were built using the following training Trials: 1) without filtering, 2) by eliminating suspicious instances, 3) by relabeling or else eliminating suspicious instances. The general observation drawn from the analysis showed that starting from 4% noise removal level and onward, using the filtering approach yielded better performance in 9 out of 10 of the domains data-sets.

4.2.3 Attribute Noise Handling Research

Khoshgoftaar et al. [41] presented a rule-based approach that detects noisy observations using Boolean rules. Observations that are detected as noisy are removed from the data before training. The approach was compared for efficiency and effectiveness against the C4.5 consensus filter algorithm presented in [15]. The results drawn from the case study suggests that when seeding noise in 1 to 11 attributes at two noise levels, the consensus filter outperforms the rule-based approach. Conversely, the rule-based approach outperforms the other approach with respect to efficiency.

Khoshgoftaar et al. [16] proposed an approach that computes noise ranks of observations relative to a user defined attribute of interest. A case study for evaluating the approach was conducted on data derived from a software project written in C and consists of 10,883 modules. In their study, the attribute of interest was chosen to be the class attribute. A comparison between the efficiency and effectiveness of the method in detecting noise and a popular classification filter algorithm [15] was made. The results reported different effectiveness scores ranging from 24% to 100% effectiveness.

Khoshgoftaar et al. [102] extended their work in [13] and proposed an approach that identifies noisy attributes in the data. Upon identifying attributes that are least correlated with the target class, those attributes get eliminated from the analysis. The approach is based on the Kendall's Tau rank correlation to identify weakly correlated attributes with the target attribute. In terms of evaluation, the effectiveness of the technique was studied using two data-sets belonging to assurance software projects, where an inspection of a software engineering expert was done to judge the performance of the approach. The overall results suggest a strong match between the output of the approach and the observations drawn from an expert in the field.

Teng [103] studied the effectiveness of three noise handling approaches, namely robustness, filtering, and correction using decision trees built by C4.5. Twelve machine learning data sets were used for the evaluation. The classification accuracy of the learners suggest that elimination and correction are viable mechanisms for minimizing the negative impact of noise. In particular, using an elimination approach before building a classifier reported an accuracy score that ranged from 77% to 100%.

Quinlan [104] demonstrated that as the noise level in the data increases, removing attribute noise information from the data decreases the predictive performance of inductive learners if the same attribute noise is present in other attributes in the data to be classified. Similarly, Zhu and Wu concluded, following a number of experiments, that attribute noise is not as harmful as class noise on the predictive performance of ML models [54].

While the majority of these work emphasize on the importance of handling both class and attribute noise in data for improving the predictive performance, the results from our study provide counter-evidence that opposes these findings when it comes to attribute noise. More precisely, the analysis results demonstrate that removing training observations that come with high attribute noise has no effect on the predictive performance of an ML classifier. These results are in line with the findings drawn by Quinlan, Brodley and Friedl, and Zhu and Wu [104], [21], [54].

4.3 Background, Definitions, and Examples

This section presents the core concepts needed to facilitate the reading of the paper. It also describes the TCS method used for the evaluation of the study, and provides definitions and examples on class and attribute noise in code churns data.

4.3.1 Core Concepts

In our approach, we use the definition of a software program P to be a collection of functions $F <F_1, \ldots, F_n>$. Each function in P consists of a sequence of statements $S <S_1,\ldots,S_n>$. *P'* denotes a modified reversion of P, and includes one or more combinations of added, removed, modified statements distant from P. In the work here, we use the term ‘old revision’ to refer to P and ‘new revision’ to refer to *P'*. The amount of code changes between P and *P'* is denoted as *code churn* and consists of a one or more statements $S <S_1,\ldots,S_n>$. A test case, denoted by *tc*, is a specification of the inputs and expected results that defines a single test to verify that *P'* complies with a specific requirement. The result of executing a single test case *tc* is referred to as ‘test case verdict’ (passed or failed) and is denoted with *tcv*. The value of *tcv* changes depending on the code against which *tc* was executed. The execution of tc is denoted with *tce*.

In this study, we use the *tcv* value of one *tce* to label each S_x in the analyzed *code churn*. A set of test cases $T = <tc_1, tc_2, \ldots>$ is the test suite for testing *P'*. Regression test selection refers to the strategy of testing *P'* using a subset of available *tc* in *T*. A duplicate entry, denoted as *de*, is the appearance of two or more combinations of syntactically identical *S* in one or more *code churns*. A set of *de* has contradictory entry if one or more combinations of *de* in the set are labeled with different test verdicts. Pairs of contradictory entries are treated as class noise.

4.3.2 Method Using Bag of Words For Test Case Selection (MeBoTS)

MeBoTS is a machine learning based method that functions as a predictor of test case verdicts [78]. The method employs a predictive model that learns from historical code churns and their relevant test case verdicts for predicting lines that would trigger test case failures. The method is described in 3 steps.

Code Changes Extractor (Step 1) The first step involves collecting code churns from designated source code repositories. To automate the collection process, we implemented a program that takes a time ordered list of historical test case execution results queried from a database. Each element in the list represents a metadata state of a previously executed test case, containing a hash reference that points at a specific location in Git's history. The program then performs a file comparison utility (diff) between pairs of consecutive hash references to extract a corpora of code churns between different revisions. All empty lines that exist in the extracted code churns are filtered out from the data before they are passed to the second step of the processing pipeline in MeBoTs.

Textual Analysis and Features Extraction (Step 2) The second step in the method involves transforming the collected code changes into feature vectors. For this purpose we used an open source tool [42] that utilizes the Bag of Words (BoW) approach for modelling textual input. The tool uses each line from the extracted code churns in step 1 and:

- creates a vocabulary for all LOC (using the bag of words technique, with a cut-off parameter of how many words should be included[1])
- creates a token for words that fall outside of the frequency defined by the cut-off parameter of the bag of words
- finds a set of predefined keywords in each line
- checks each word in the line to decide if it should be tokenized or if it is a predefined feature

[1]BoW is essentially a sequence of words/tokens, which are descendingly ordered according to frequency. This cut-off parameters controls how many of the most frequently used words are included as features – e.g. 10 means that the 10 most frequently used words become features and the rest are ignored.

Table 4.2: Comparing Popular Defect and Test Prediction Approaches with MeBoTs.

Method	**Description**	**Pros and Cons**	**MeBoTs**
Code metrics	Uses code static metrics, such as code complexity, size, churn metrics to train machine learning models on classifying defective code. Examples: [105], [106]	**Pros:** - Strong empirical evidence that supports the use of some code metrics for defects prediction for Java programs. **Cons:** - Static metrics need to be decided a priori, and they depend on the size.	- language agnostic and can be applied on any programming language. - The features from MeBoTS are not decided a priori, and are not dependent on size
Static Code Analysis	Uses machine learning models to learn semantic features derived from abstract syntax trees. Examples: [96], [97], [107]	**Pros:** - Characterize defects using abstract syntax tree information from the code. **Cons:** - Code needs to be compiled. - Does not scale well when the number of tree nodes increases.	- Generates feature vectors from the actual program using textual analysis -Does not require the code to be compiled. - Uses statistics to generate its feature vectors.
Dynamic Analysis	This category relies on executing the program and comparing its actual with expected behavior. Examples: [108], [109]	**Pros:** -Allows for analysis of the program without having access to the code. **Cons:** - If the code does not run, no analysis is done	- Analyzes the code before compiling the program.

Therefore, MeBoTs treats code tokens as features and represents a code line with respect to its tokens' frequencies. To our knowledge, this way of extracting feature vectors from the source code is new in our approach, compared with other popular approaches for defects and test prediction. In particular, MeBoTS can directly recognize what is written in the code without the need to compile the code and access its abstract syntax tree for generating feature vectors. Table 4.2 lists and describes some of the most popular approaches for defects and test prediction using source code analysis. It also highlights a few advantages and disadvantages of these approaches and contrasts them with MeBoTs.

Training and Applying the Classifier Algorithm (Step 3) We exploit the set of extracted features provided by the textual analyzer in step 2 and the verdict of the executed test cases for training a predictive model on classifying LOC into either triggering to test case failure or not.

4.3.3 Noise Definitions and Examples

Noisy observations are commonly determined by two factors: 1) the correctness of the class values, and 2) by how well the selected attributes describe learning instances in the training data. This section provides a definition and an example for each type of noise (class and attribute) found when analyzing input data that corresponds to code churns (attributes) and *tcv* (class).

4.3.3.1 Example of the dependency between code churns and test case verdict

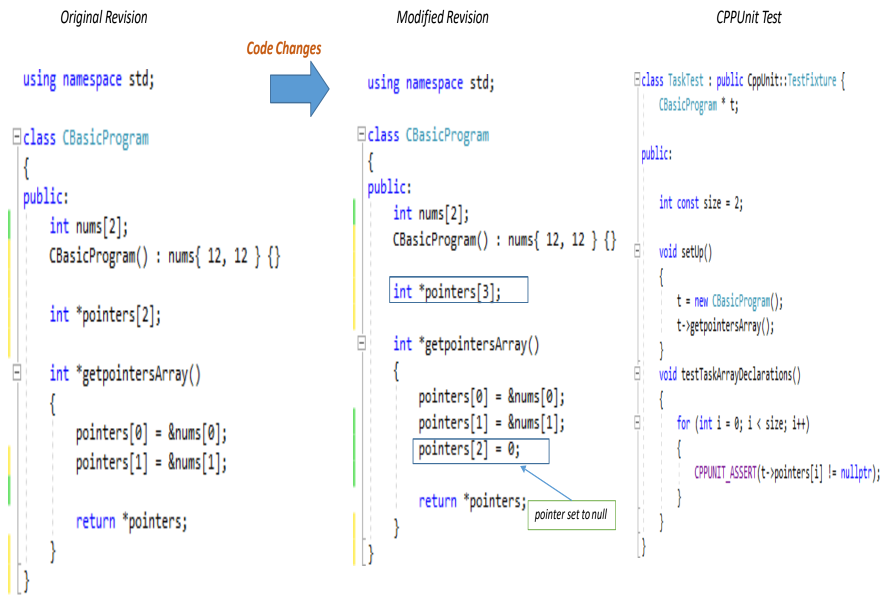
In this subsection, we present an example that illustrates the dependency between code churns and test case verdicts. The example shows how a unit test case will react to a code change in *P'* of *P*. Figure 4.2 shows two revisions of an example program *P* written in the C++ language. The modified revision *P'* in the Figure includes the same code fragments in *P* except for the two framed statements *S1 and S2*. *S1* is a declaration of an array of type `int*`, whereas *S2* is an assignment of value 0 to the array element `pointers[2]` in `F1:getpointersArray`. In the C++ language, pointers that are assigned the value of 0 are called *null pointers* because a memory location of address 0 does not exist and therefore a run-time exception will be thrown when executing the program. To avoid such assignments in the code base, a unit test case `tc1:testTaskArrayDeclarations` is created to assert that all elements in the pointers' array are not set to null (assigned 0), as shown in Figure 4.2. By executing *tc1* against *P'*, we observe from the that the code churn *S1 and S2* triggered the *tcv* of *tc1* to change from a Pass to a Failing state. The reaction of tc1 to the churned P showcases the dependency between code churns and test case verdict. Therefore, the underlying theory that test cases would react to code churns is worth exploring for predicting test case verdicts for test case selection.

4.3.3.2 Definition and Example of Class Noise in Code Churns Data

In this study, class noise is defined as the ratio of contradictory entries *de* to the overall number of entries in the analyzed data. Since a contradictory entry can only occur among two (or more) *de*, the number of all duplicate entries for which an entry is assigned a different class label is identified as a contradictory entry. More formally, the formula for calculating this noise ratio can be expressed as follow:

$$\text{Class Noise ratio} = \frac{\text{Number of Contradictory Entries}}{\text{Total Number of Entries}}$$

For example, a data-set containing six *de* with five *de* labeled as `true` and one labeled as `false` has six contradictory entries. Finding a rule to identify which class should be used to correct a mislabelled entry is not trivial, since we do not know the context in which these entries occurred nor the sources of noise that triggered the differential class labels.



```
using namespace std;

class CBasicProgram
{
public:
    int nums[2];
    CBasicProgram() : nums{ 12, 12 } {}

    int *pointers[2];

    int *getpointersArray()
    {
        pointers[0] = &nums[0];
        pointers[1] = &nums[1];

        return *pointers;
    }
}
```

```
using namespace std;

class CBasicProgram
{
public:
    int nums[2];
    CBasicProgram() : nums{ 12, 12 } {}

    int *pointers[3];

    int *getpointersArray()
    {
        pointers[0] = &nums[0];
        pointers[1] = &nums[1];
        pointers[2] = 0;

        return *pointers;
    }
}
```

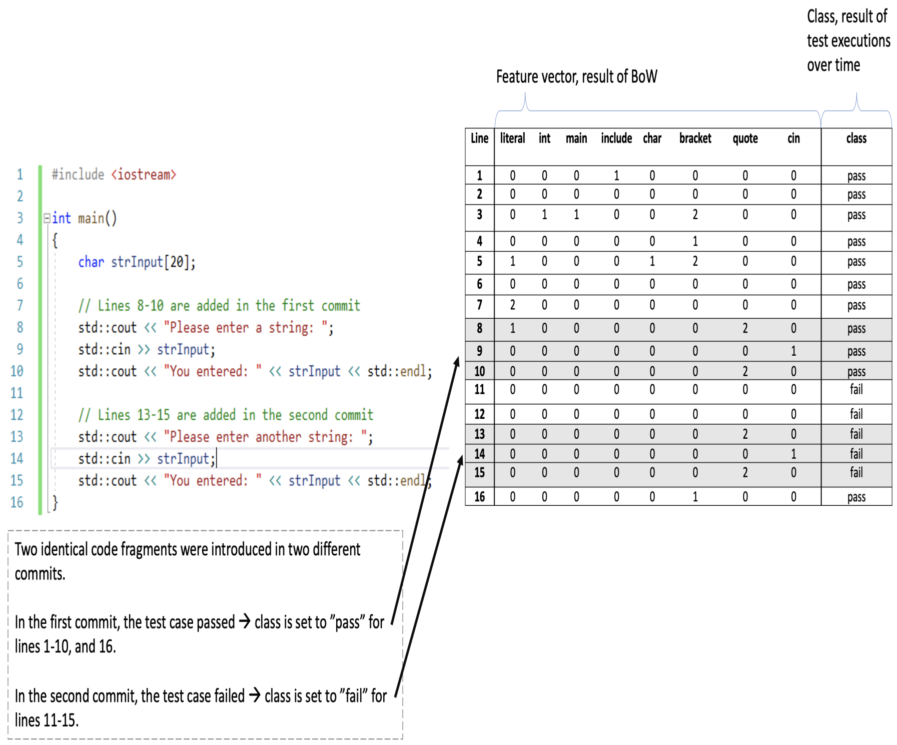
```
class TaskTest : public CppUnit::TestFixture {
    CBasicProgram * t;

public:

    int const size = 2;

    void setUp()
    {
        t = new CBasicProgram();
        t->getpointersArray();
    }
    void testTaskArrayDeclarations()
    {
        for (int i = 0; i < size; i++)
        {
            CPPUNIT_ASSERT(t->pointers[i] != nullptr);
        }
    }
}
```

Figure 4.2: Example On the Relationship Between Code Churns and Test Case Verdicts

As an illustration of the class noise problem in a data-set consisting of code churns, Figure 4.3 shows a sample C++ program transformed into feature vectors using the BoW approach. Each line of code in the sample program is transformed into a line vector which gets assigned a class value based on a *tce* result for the committed code. These transformed lines and their relevant *tce* get fed as input into an ML model for training. The model is used to predict which lines in the program will trigger a test case failure or success.

The feature vectors in Figure 4.3 characterize code lines in the sample program. All shaded lines in the sparse matrix (lines 8, 9, 10, 13, 14, and 15) are contradictory entries since each of the pairs (8 and 13), (9 and 14), and (10 and 15) have the same vectors but different class values (pass and fail). The formula for calculating the class noise ratio in this example is:

$$\text{Class Noise ratio} = \frac{6}{16} = 0.375$$


Feature vector, result of BoW
Class, result of test executions over time

```
1   #include <iostream>
2
3   int main()
4   {
5       char strInput[20];
6
7       // Lines 8-10 are added in the first commit
8       std::cout << "Please enter a string: ";
9       std::cin >> strInput;
10      std::cout << "You entered: " << strInput << std::endl;
11
12      // Lines 13-15 are added in the second commit
13      std::cout << "Please enter another string: ";
14      std::cin >> strInput;
15      std::cout << "You entered: " << strInput << std::endl;
16  }
```

Line	literal	int	main	include	char	bracket	quote	cin	class
1	0	0	0	1	0	0	0	0	pass
2	0	0	0	0	0	0	0	0	pass
3	0	1	1	0	0	2	0	0	pass
4	0	0	0	0	0	1	0	0	pass
5	1	0	0	0	1	2	0	0	pass
6	0	0	0	0	0	0	0	0	pass
7	2	0	0	0	0	0	0	0	pass
8	1	0	0	0	0	0	2	0	pass
9	0	0	0	0	0	0	0	1	pass
10	0	0	0	0	0	0	2	0	pass
11	0	0	0	0	0	0	0	0	fail
12	0	0	0	0	0	0	0	0	fail
13	0	0	0	0	0	0	2	0	fail
14	0	0	0	0	0	0	0	1	fail
15	0	0	0	0	0	0	2	0	fail
16	0	0	0	0	0	1	0	0	pass

Two identical code fragments were introduced in two different commits.

In the first commit, the test case passed → class is set to "pass" for lines 1-10, and 16.

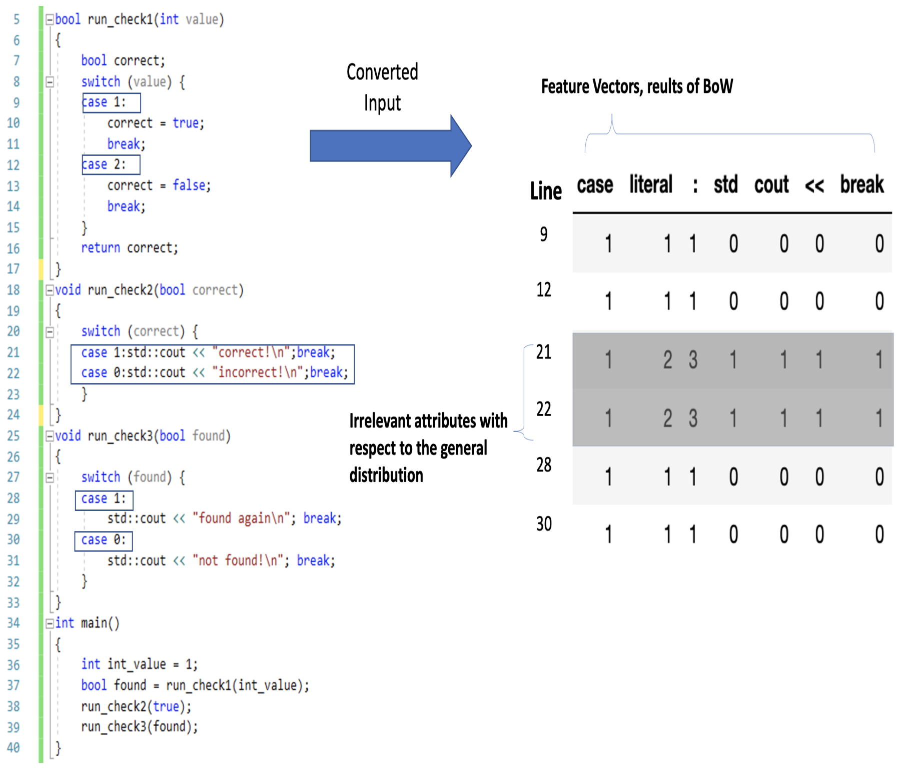
In the second commit, the test case failed → class is set to "fail" for lines 11-15.

Figure 4.3: Class Noise Example in Code Base.

4.3.4 Definition and Example of Attribute Noise in Code Churns Data

The definition of attribute noise in this paper follows the one proposed by Van Hulse et al. [13], which suggests that a noisy observation appears when one or more of its attributes deviates from the general distribution of other attributes. The larger the deviation is for one or more observations, the more evidence there is that they are noisy. In the context of the given problem (i.e., TCS), a deviation between attributes can occur when the general distribution of S follows a standard coding style, whereas a smaller fraction of S deviates from the standard.

As an illustration of those deviations in code churns, Figure 4.4 exemplifies two coding styles used for expressing `case` blocks in a C++ program. By examining the `case` blocks in the `run_check1`, `run_check2`, and `run_check3` functions, we notice that the first and most reoccurring style uses a line space to separate statements in a `case` block, as shown in the `run_check1` and `run_check3` methods. Conversely, the other coding style used in `run_check2` aligns all set of S in a case block on one line. The attributes in this example are feature vectors that correspond to tokens in the code fragment. Note how $S21$ and $S22$ are characterized by additional attribute that deviate from the majority of attributes in the remaining `case` blocks at *S9, S12, S28, and S30*. Those deviations in S21 and S22 from the rest `case` statements are considered suspicious and therefore irrelevant.



Converted
Input
Feature Vectors, reults of BoW
Irrelevant attributes with respect to the general distribution

```
5   bool run_check1(int value)
6   {
7       bool correct;
8       switch (value) {
9       case 1:
10          correct = true;
11          break;
12      case 2:
13          correct = false;
14          break;
15      }
16      return correct;
17  }
18  void run_check2(bool correct)
19  {
20      switch (correct) {
21      case 1:std::cout << "correct!\n";break;
22      case 0:std::cout << "incorrect!\n";break;
23      }
24  }
25  void run_check3(bool found)
26  {
27      switch (found) {
28      case 1:
29          std::cout << "found again\n"; break;
30      case 0:
31          std::cout << "not found!\n"; break;
32      }
33  }
34  int main()
35  {
36      int int_value = 1;
37      bool found = run_check1(int_value);
38      run_check2(true);
39      run_check3(found);
40  }
```

Line	case	literal	:	std	cout	<<	break
9	1	1	1	0	0	0	0
12	1	1	1	0	0	0	0
21	1	2	3	1	1	1	1
22	1	2	3	1	1	1	1
28	1	1	1	0	0	0	0
30	1	1	1	0	0	0	0

Figure 4.4: Attribute Noise Example in Code Base.

4.4 Noise Handling and Removal Approaches

The problem of achieving a good learning performance in the presence of noisy environments has been widely highlighted in the ML literature. Several approaches have been built to enhance the learning performance of ML classifiers [84], [54], [40]. Nevertheless, the presence of class and attribute noise have been reported to still have a negative influence on the learning, and thus needs to be handled before training. In this section, we describe an approach that we introduced in the baseline study [98] to handle the problem of class noise. Thereafter, an existing elimination based approach from the literature for handling attribute noise is described.

4.4.1 Class Noise Approach

Our approach for handling annotation noise relies on relabeling repeated code lines that come with different class values. These repeated lines can potentially occur in code churns due to several scenarios, such as 1) copying of code [110], and 2) merge transactions [99]. The first scenario manifests itself in the event of 'copy-paste' reuse of code check-ins that had previously passed the testing and integration phases. In such scenario, the developers explicitly duplicate source code fragments to adapt the duplicates for a new purpose in a quick fashion [110]. The second scenario appears when developers in one or more teams work on dedicated branches for features development and use similar code phrases as to those committed and merged from different branches [99] e.g., `x = x + 1;`. When extracting such code check-ins with duplicate code phrases for TCS, inconsistent observations with different class values might occur.

To address the problem of contradictory code lines in code churns data, we present an approach that relies on domain knowledge for identifying instances (code lines) that require relabeling. We use the phrase *class-noise cleaned* to refer to a data-set on which the class noise handling approach was applied. A step-by-step description of the approach is as follow:

- sequentially assign a unique 8-digit hash value for each line of code in the original data set
- create an empty dictionary for storing unfiltered lines of code.
- iterate through the set of hashed lines in the original data set and save non-repeated (syntactically unique) lines of code in the dictionary.
- compare the annotation values of each pair of `duplicate lines` in the original and dictionary sets. If the annotation value of the repeated instance in the original set is annotated with 1 (passed) and the annotation value of the same instance in the dictionary is annotated with 0 (failed), then relabel the annotation value for the instance in the dictionary from 0 to 1. If the annotation values of both `duplicate lines` are annotated with '1' then skip adding the entry from the original set into the dictionary.

This way of handling annotation noise can be seen as both corrective and eliminating, since it 1) corrects the label of duplicate entries that first appears as failing and then pass the test execution, and 2) removes duplicate lines that are labeled as passing.

Defective lines often occupy a small proportion of the overall fragment of code changes. Thus, a random line from a fragment, which was overall labeled as failing is more likely not to be the cause of the failure. Therefore, our design decision is to relabel lines as 'passed', if they have already been seen as part of non-failing fragments before. Thus, we select a more conservative approach when it comes to labeling lines as failing, in order to minimize the likelihood of mislabeling training entries[2].

[2]https://github.com/khaledwalidsabbagh/Annotation_Noise

4.4.2 Selected Attribute Noise Handling Approach

As mentioned earlier, attribute noise can occur due to selecting attributes that are irrelevant for characterizing the training instances. In the domain of TCS, those attributes can materialize when, for example, the analyzed code consists of fragments that are written using different coding styles or when a small number of statements/conditions/function declarations etc deviate in syntax from the majority of similar lines in the code.

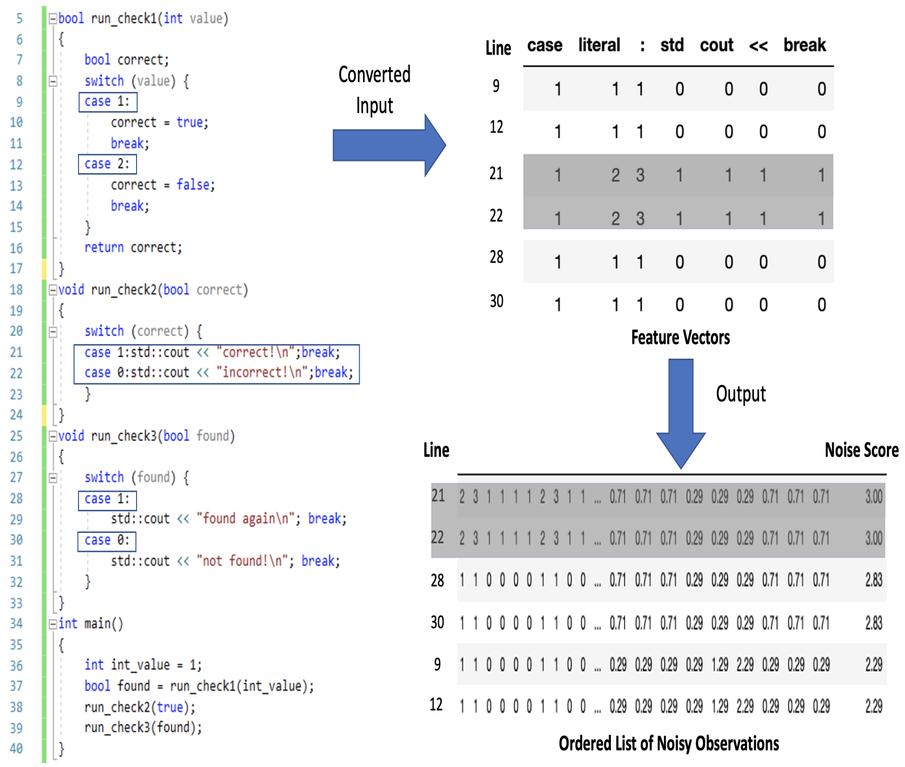
To address the problem of attribute noise in training data, we decided to use an existing elimination based approach called PANDA [13] that identifies training instances with large deviations from normal. The PANDA algorithm identifies such instances by comparing pairs of attributes in the space of feature vectors. The output is an ordered list of noise scores for each code line - the higher the noise score for a code line, the higher it deviates from normal. Upon ranking noisy instances, the generated list can be used to toss out instances (code lines) that come with the highest rank with respect to attribute noise.

The algorithm starts by iterating through all attributes in the input feature vectors. In each iteration, a single attribute x_j gets partitioned into a number of bins, based on a predefined bin value that is set by the user. Each bin will have the same amount of instances, given that the number of input observations is divisible by the number of partitions. In the absence of tied values, the algorithm includes all boundary instances that fall outside the range of the bin size in the last bin. After the partitioning is complete, the mean and standard deviation for instances in each bin are calculated and used to derive a standardized value for each instance in attribute x_k. The standardized value is then calculated by subtracting the ratio of mean to standard deviation in the bin relative to x_j from each instance value in x_k. This approach is repeated for all attributes in the input space of vectors. Finally the MAX or the SUM value of each observation is calculated. Large sum or max values indicate an observation that has a high attribute noise value.

Figure 4.5 exemplifies the output produced by the PANDA algorithm when applied on the code fragment presented in Section 4.3.4. Note that in this example, only lines that start with the keyword 'case' were input to the algorithm, whereas in our experiment, all code lines in the sample data-set were input. The bins' size in the example program was set to 1 and the output produced is a list of observations ordered from the most noisy to the least noisy using the MAX function. Note that the highest noise scores in the sample data were identified for lines 21 and 22 as their attribute values deviate from the remaining majority of the 'case' statements in lines 9, 12, 28, and 30.

4.5 Research Methodology

The goal of this paper is to examine the effect of handling class and attribute noise in code change data-sets for improving test case selection. This section describes the design and operations carried out for analyzing the impact of class and attribute noise handling on the predictive performance of a learner



```
5   bool run_check1(int value)
6   {
7       bool correct;
8       switch (value) {
9       case 1:
10          correct = true;
11          break;
12      case 2:
13          correct = false;
14          break;
15      }
16      return correct;
17  }
18  void run_check2(bool correct)
19  {
20      switch (correct) {
21      case 1:std::cout << "correct!\n";break;
22      case 0:std::cout << "incorrect!\n";break;
23      }
24  }
25  void run_check3(bool found)
26  {
27      switch (found) {
28      case 1:
29          std::cout << "found again\n"; break;
30      case 0:
31          std::cout << "not found!\n"; break;
32      }
33  }
34  int main()
35  {
36      int int_value = 1;
37      bool found = run_check1(int_value);
38      run_check2(true);
39      run_check3(found);
40  }
```

Line	case	literal	:	std	cout	<<	break
9	1	1	1	0	0	0	0
12	1	1	1	0	0	0	0
21	1	2	3	1	1	1	1
22	1	2	3	1	1	1	1
28	1	1	1	0	0	0	0
30	1	1	1	0	0	0	0

Line																					Noise Score
21	2	3	1	1	1	1	2	3	1	1	...	0.71	0.71	0.71	0.29	0.29	0.29	0.71	0.71	0.71	3.00
22	2	3	1	1	1	1	2	3	1	1	...	0.71	0.71	0.71	0.29	0.29	0.29	0.71	0.71	0.71	3.00
28	1	1	0	0	0	0	1	1	0	0	...	0.71	0.71	0.71	0.29	0.29	0.29	0.71	0.71	0.71	2.83
30	1	1	0	0	0	0	1	1	0	0	...	0.71	0.71	0.71	0.29	0.29	0.29	0.71	0.71	0.71	2.83
9	1	1	0	0	0	0	1	1	0	0	...	0.29	0.29	0.29	0.29	1.29	2.29	0.29	0.29	0.29	2.29
12	1	1	0	0	0	0	1	1	0	0	...	0.29	0.29	0.29	0.29	1.29	2.29	0.29	0.29	0.29	2.29

Figure 4.5: An Excerpt of PANDA's Output

for test selection.

4.5.1 Original Data Set

In the baseline paper [98], we worked with a data set of code churns that belong to a legacy system written in the C language. A total of 82 test case execution results (35 passed tests and 47 failed tests) for 12 test cases and their relevant set of code changes (1.4 million LOC) were collected. The system from which the sample data was extracted belongs to a large Swedish telecommunication company and has the size of several million lines of code. The feature vectors generated from the data-set in [98] using a bi-gram BoW model comprised a total of 2251 features/attributes. The distribution of the binary classes in the collected data was fairly balanced, with 44% of the code lines belonging to the 'passed' class and 56% to the 'failed' class [3].

[3]Due to non-disclosure agreements with our industrial partner, our data-set can not be made public for replication.

4.5.2 Random Forest For Evaluation

In this study, the MeBoTS method described in Section 4.3 was used as an example of a TCS approach. The selected learning model for the evaluation was random forest (RF), mainly due to its low computational cost and white-box nature compared with deep learning models. In the context of MeBoTS, using a white-box model, such as RF, is important since it can showcase the feature importance charts. These charts can provide practitioners with insights into the tokens that led to the prediction of failing test cases.

The hyper-parameters of the model were kept in their default state as found in the scikit-learn library (version 0.20.4). The only configuration made was in the n_estimator (the number of trees) parameter, where we changed it from 10 to 100. We did not experimentally seek to tune the n_estimater value in the RF model, since the goal of this study is not to optimize the predictive performance of the model, but rather to examine the effect of attribute and class noise on TCS. However, we experimented the use of another variation of the n_estimater in the RF model (n_estimater=300) in order to get an understanding of whether this would affect the model's predictive performance. The performance results produced by the model with 300 trees can be found in Appendix A.

4.5.3 Class Noise

The evaluation of the presented class noise approach was done by comparing the learning performance of the ML model in MeBoTS under two training trials 1) using the original (uncleaned) data, and 2) using a class-noise cleaned data. For each training trial, we measured the performance in terms of precision, recall, and F1-score, for an ML model.

Applying the class noise handling approach (described in Section 4.4.1) on the original (uncleaned) data-set resulted in a reduced set, comprising of 140,130 LOC. We use the adjective 'class-noise cleaned' to refer to this reduced set. The number of lines labelled as passing in the cleaned set were 46%, whereas the remaining 54% of the lines were labelled as failing. A random split of the class-noise cleaned data was performed to generate s training and testing sets. The size of the training set comprised of 112,104 line vectors, whereas the remaining 28,026 line vectors were used for evaluating the learning of the model.

4.5.4 Attribute Noise

The extension provided in the study focuses on examining the effect of eliminating instances with attribute noise on the learning performance for TCS. To identify possible causality between attribute noise and learning performance, a controlled experiment was carried out. This subsection describes the experimental design and operations conducted to examine the causality.

4.5.4.1 Adopted Data-Set

In this study we wanted to get an initial understanding of the effect of attribute noise on the learning performance of an ML model for TCS. Therefore, we experimented the effect of attribute noise removal on a subset of observations and attributes from the class-noise cleaned data. The selected subset was created by randomly selecting 19,815 instances and 800 attributes. This data-set will act as the control group and will be used as a baseline for class-noise cleaned data.

According to Ganganwar and Vaishali [111], a data-set is called imbalanced when it contains many more samples under one class than from the rest of the classes. Accordingly, we inspected the distribution of the samples in the control group with respect to the binary classes (defective and non-defective) in order to determine the balance of classes. Figure 4.6 shows that the distribution of instances in the non-defective class contains many more samples than the defective class (14,400 to 5,415 samples). Based on this distribution and given that we only have two classes (binomial distribution), we consider the control group to be imbalanced. To overcome this problem, we chose to upsample instances in the minority class using the 'resample' module provided in the Scikit-learn library [76]. The idea of oversampling is to randomly generate samples from the minority class instances until the number of minority class is the same as the number of majority class.

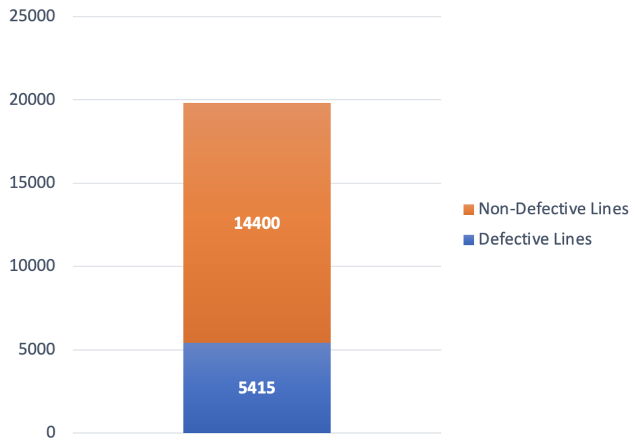


Figure 4.6: The Distribution of Classes In The Adopted Data-Set

4.5.4.2 Independent Variable and Experimental Subjects

In this study, attribute noise removal was the only independent variable (treatment) examined for an effect on classification performance. Ten variations of the treatment were selected. Namely, 5%, 10%, 15%, 20%, 25%, 30%, 35%, 40%, 45%, 50%. Each treatment level corresponds to a fraction size of observations that gets eliminated before training the ML model in MeBoTS. We used

25-fold stratified cross validation on the control group to derive 25 experimental subjects on which the treatment is applied. Each subject is treated as a hold out group for validating an RF model which gets trained on the remaining 24 training subjects. A total of 275 trials derived from the 25-folds were conducted - each 25 trials for evaluating the performances of a learner under one treatment level.

4.5.4.3 Dependent Variables

The dependent variables are three evaluation measures used for the performance of an ML classifier – Precision, Recall, and F1-score. The three evaluation measures are defined as follows:

- Precision is the fraction of passing-classified tests that are actually passing.
- Recall is the fraction of really passing tests that are classified as passing.
- The F1-score is a harmonic mean between precision and recall.

When the precision of a classifier is high, less test cases that do not detect faults in the system under test are executed, whereas when the recall is high less false alarms about detected faults are produced. Therefore, the higher the precision and recall a classifier gets, the better the test selection process.

4.5.4.4 Experimental Hypotheses

Three hypotheses are defined according to the goals of this study and tested for statistical significance in Section4.5.4.5. The hypotheses were based on the assumption that data-sets with more attribute noise have a significantly negative impact on the classification performance of an ML model for TCS compared to data with no attribute noise. The hypotheses are as follow:

- *H0p: The mean Precision is the same for a model with and without attribute noise*
$$\mu 1p = \mu 2p \tag{4.1}$$
- *H0r: The mean Recall is the same for a model with and without attribute noise*
$$\mu 1r = \mu 2r \tag{4.2}$$
- *H0f: The mean F1-score is the same for a model with and without attribute noise*
$$\mu 1f = \mu 2f \tag{4.3}$$

For example, the first hypothesis can be interpreted as: *a data-set with a higher attribute noise ratio will result in significantly lower Precision rate, as indicated by the mean Precision score across the experimental subjects.* After evaluating the hypotheses, we compare the evaluation measures under each treatment level with those at 0% attribute noise removal level.

4.5.4.5 Data Analysis Methods

The experimental data were analyzed using the scikit learn library [76]. To decide whether to use a parametric or non-parametric test for the analysis, a normality test was carried out. First, we plotted the frequency distribution graphs for the three dependent variables under each treatment level to see if they deviate from a normal distribution. To further validate the visual inspection, a Shapiro-Wilk test was carried out. The results showed that 3 dependent variables are not normally distributed (see Section 4.6.2 for details). Based on the normality test results, we decided to use two non-parametric tests, namely: Kruskal-Wallis and Mann-Whitney. To evaluate the hypotheses, the Kruskal-Wallis was selected for comparing the median scores between the three evaluation measures under the 11 treatment levels. The Mann–Whitney U test was selected to perform a pairwise comparison between the evaluation measures under each treatment level and the same measures with no treatment (0% noise removal).

4.5.4.6 Attribute Noise Removal

As mentioned earlier, the adopted data-set acts as the control group in this experiment. This control group is used to examine the effect of the treatment on the learning performance of the ML model in MeBoTS (RF). Moreover, we use this group as a baseline for comparing the effect of class noise handling and the attribute noise removal approaches on learning.

To apply the treatment, we began by running the PANDA algorithm on the control group. The output is an ordered list of observations that are ranked with respect to the amount of noise identified in their attributes. Table 4.3 shows an excerpt of the three top ranked observations generated in the ordered list. Note that due to the non-disclosure agreement with our industrial partner, all original keywords in the 'Code Line' column, such as variable and class names, are replaced with artificial variable names. The indexes in the first column of the list are used to retrieve and eliminate a fraction of observations from the training subjects. The size of the fraction depends on the desired treatment level. For instance, a treatment of 5% implies retrieving 5% of observations that are top ranked in the PANDA's list (5% of 19,815 LOC) and from the training subjects and removing them. In this experiment, ten variations of the treatment was applied (5%, 10%, 15%, 20%, 25%, 30%, 35%, 40%, 45%, and 50%). For each treatment, a fraction of observations that is equivalent in ratio to the treatment level is fetched and removed from the training subjects. As soon as those observations are removed, the training subjects are fed into an ML model for training and the precision, recall, and F1-score are recorded for the model.

In this experiment, a bin size of five was used in the PANDA. This means that each attribute in the analyzed data is split into five bins and the comparison between each pair of attributes is done relative to those bins. The implementation of the PANDA algorithm used in this study can be found at the link in the footnote[4].

Table 4.3: An Excerpt Of the Output Generated From PANDA

Index	Code Line	Noise Score
1181	__class__ ((constructor))	518
1056	if (!isNotEmpty() && sharedPool)	518
1051	// addPoolConfig return value	518

4.6 Evaluation Results

In this section, we present and compare the results of learning obtained from training on 1) the original and class-noise cleaned data, and 2) the class-noise cleaned data and the class and attribute noise cleaned data-sets. We report the learning in terms of precision, recall, and F1-score.

4.6.1 Original vs. Class Noise Cleaned Data

The performance measurements of the RF classifier built on the class-noise cleaned data is plotted using a confusion matrix, as shown in Figure 4.7. The Figure shows a non-normalized matrix for the predicted and actual values of test case verdicts for all lines in the test set. The first cell on the upper left hand side corresponds to the number of lines (6,543) that are predicted to trigger test case failures and are actually true. On the same diagonal, the last cell to the bottom right of the matrix indicates the number of lines (15,688) that are predicted to be non-defective and are actually true, and require no testing. The remaining entries in the test set correspond to the number of misclassified lines.

The bar chart in Figure 4.8 illustrates the performance measures of the classifier built on the original and class-noise cleaned data. The results reveal that handling class noise in the uncleaned data improves the learning performance by 70% recall, 37% precision, and 59% F1-score compared to the learning achieved on the original data.

4.6.2 Class Noise Cleaned vs. Class and Attribute Noise Cleaned Data

This subsection discusses the results of the descriptive statistics and statistical tests conducted to evaluate hypotheses *H0p, H0r, and H0f* presented in Section 4.5.4.4. The results reported in this section are used for drawing a comparison between the effectiveness of handling class noise and attribute noise on the learning performance. Figures 4.9, 4.10, and 4.11 show three box-plot graphs, which were plotted to visually inspect the effect of removing observations with attribute noise at each treatment level on the dependent variables. A first

[4]https://github.com/khaledwalidsabbagh/Handling_Attribute_Noise_PANDA

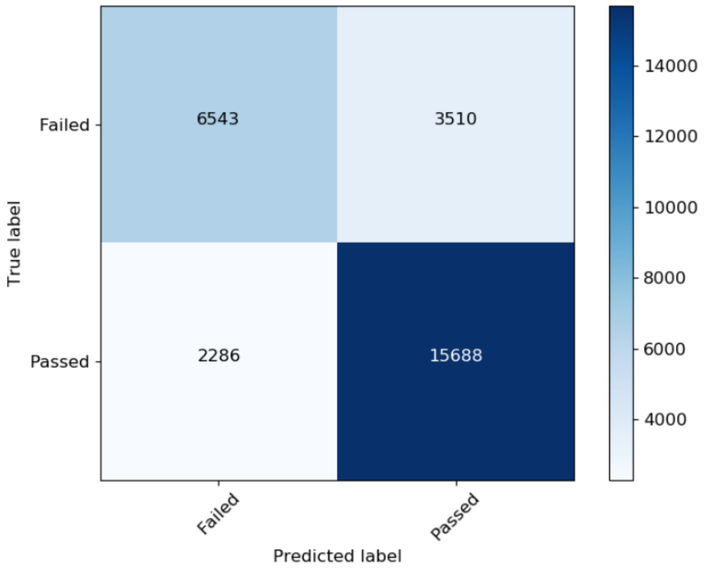


Figure 4.7: Confusion Matrix For a Classifier Trained on Class Noise cleaned Data

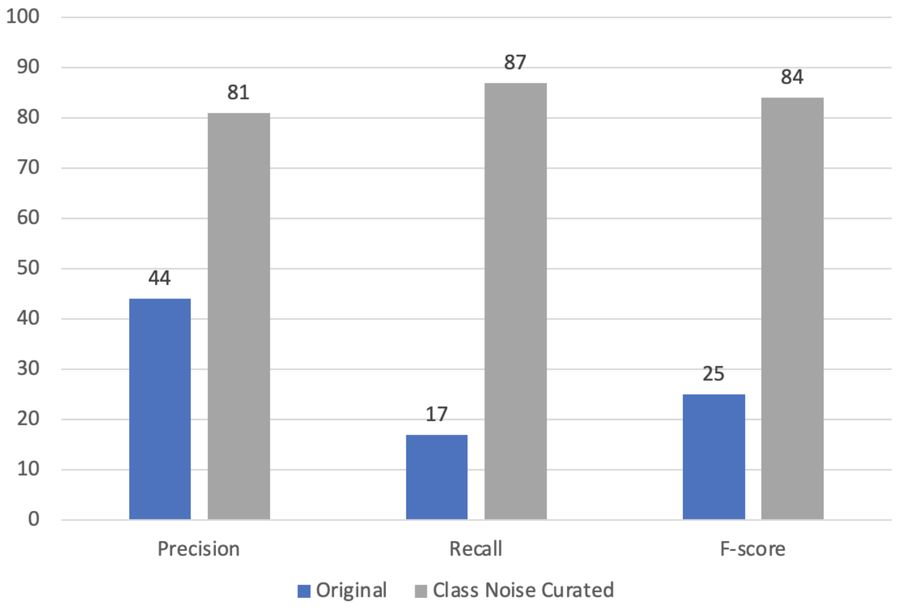


Figure 4.8: Learning Performance On the Original and the Class Noise cleaned Data Sets

observation from the graphs suggests a lack of causality between the treatment and the three dependent variables. This observation was further supported by examining the mean scores of each dependent variable in the descriptive statistics, as shown in Tables 4.4, 4.5, and 4.6. Note that the precision, recall, and F1-score reported in the three tables under 0% treatment level are different than those obtained from training on the class-noise cleaned data. This is because the control group was used as a baseline for the class-noise cleaned data from which the ML model was built.

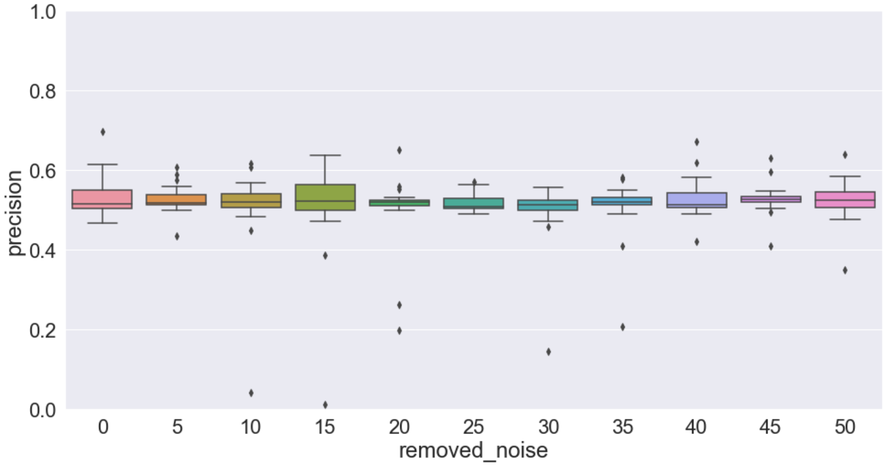


Figure 4.9: The Distribution of Precision Values Under the Treatment Levels

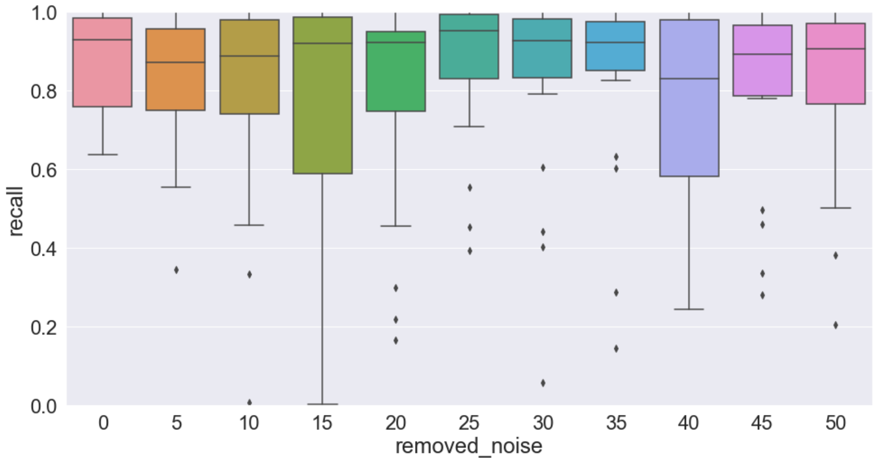


Figure 4.10: The Distribution of Recall Values Under the Treatment Levels

To begin the evaluation of the hypotheses, we start by checking the normality in the distribution of the three dependent variables. The frequency distribution

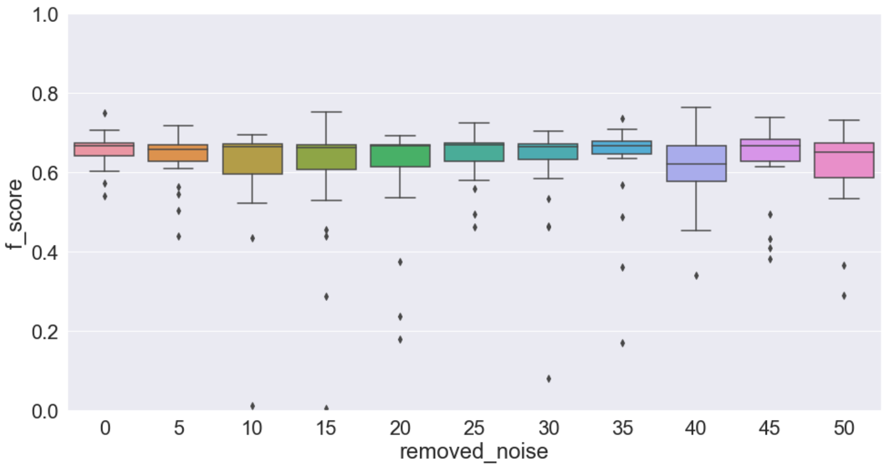


Figure 4.11: The Distribution of F1-score Values Under the Treatment Levels

Table 4.4: Descriptive Statistics For Precision.

Treatment	N	Mean	SD	SE	95% Conf.	Interval
0%	25	0.53	0.05	0.01	0.51	0.55
5%	25	0.53	0.03	0.01	0.51	0.54
10%	25	0.51	0.1	0.02	0.47	0.55
15%	25	0.51	0.12	0.02	0.46	0.56
20%	25	0.5	0.09	0.02	0.47	0.54
25%	25	0.52	0.02	0.0	0.51	0.53
30%	25	0.5	0.08	0.02	0.47	0.53
35%	25	0.51	0.07	0.01	0.48	0.54
40%	25	0.53	0.05	0.01	0.51	0.55
45%	25	0.53	0.04	0.01	0.51	0.54
50%	25	0.53	0.05	0.01	0.51	0.55

of the variables were plotted for the 275 trials (25 trials for each treatment level) to visually inspect normality, as shown in Figures 4.12, 4.13, and 4.14. Then, the Shapiro-Wilk test was carried out to further support the observations drawn from the graphs. As can be seen from the graphs, the distributions appear to be negatively skewed (asymmetric), and thereby the assumption of normality in the distribution of the three variables do not hold. The Shapiro-Wilk test results supported the observation drawn from the graphs and revealed that the null hypotheses of normality for the three dependent variables can be rejected (p-value <0.05), as shown in Tables 4.74.8. Since we have issues with normality in the samples, we decided to run a non-parametric test for comparing the difference between the performance measures under the 10 treatment levels.

To examine the effect of removing observations with top rank attribute noise on the learning, the Kruskal-Wallis test was conducted. Table 4.9 sum-

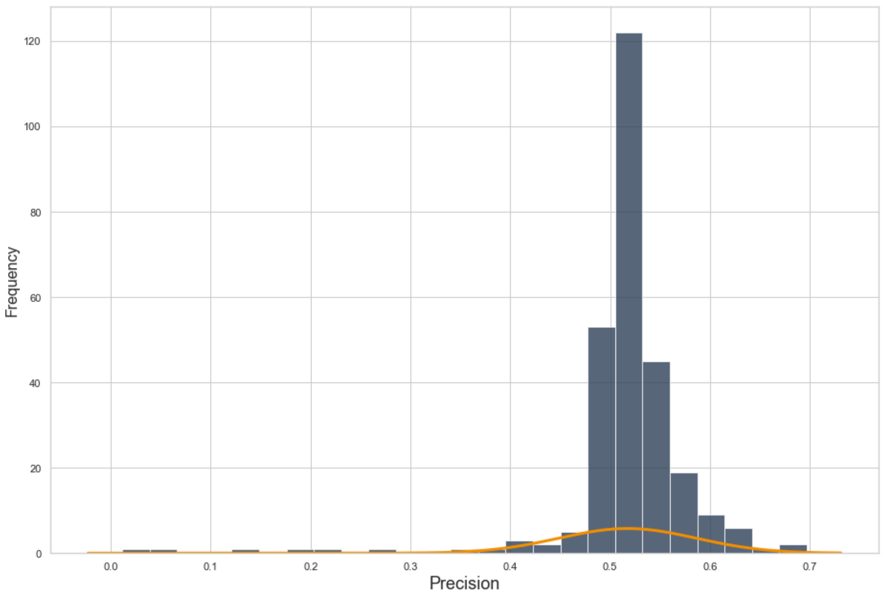


Figure 4.12: Frequency Plot For the Precision Scores

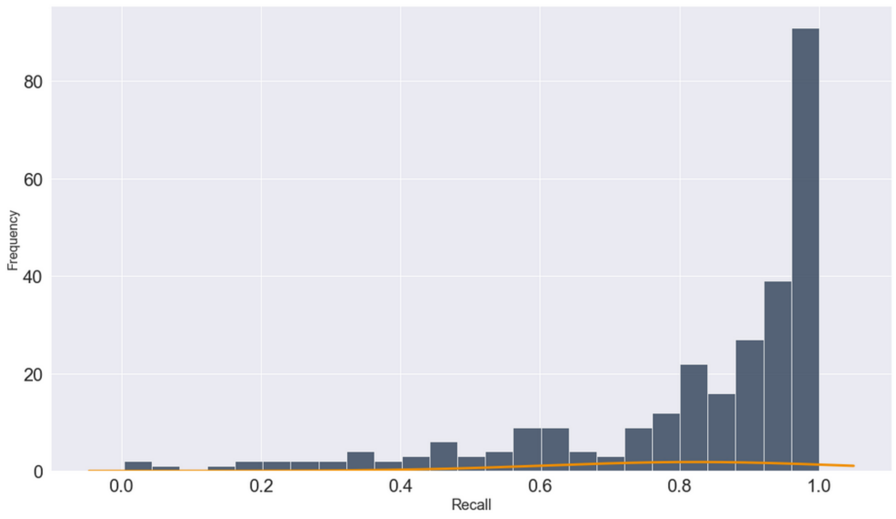


Figure 4.13: Frequency Plot For the Recall Scores

Table 4.5: Descriptive Statistics For Recall.

Treatment	**N**	**Mean**	**SD**	**SE**	**95% Conf.**	**Interval**
0%	25	0.88	0.13	0.03	0.83	0.93
5%	25	0.83	0.17	0.03	0.76	0.9
10%	25	0.8	0.25	0.05	0.7	0.9
15%	25	0.78	0.27	0.05	0.67	0.89
20%	25	0.8	0.25	0.05	0.7	0.9
25%	25	0.88	0.17	0.03	0.81	0.95
30%	25	0.84	0.23	0.05	0.75	0.93
35%	25	0.85	0.22	0.04	0.76	0.94
40%	25	0.77	0.23	0.05	0.67	0.86
45%	25	0.82	0.21	0.04	0.74	0.9
50%	25	0.8	0.22	0.04	0.72	0.89

Table 4.6: Descriptive Statistics For F1-score.

Treatment	**N**	**Mean**	**SD**	**SE**	**95% Conf.**	**Interval**
0%	25	0.66	0.04	0.01	0.64	0.67
5%	25	0.64	0.06	0.01	0.61	0.66
10%	25	0.61	0.14	0.03	0.55	0.66
15%	25	0.6	0.16	0.03	0.53	0.66
20%	25	0.61	0.14	0.03	0.55	0.66
25%	25	0.65	0.06	0.01	0.62	0.67
30%	25	0.62	0.13	0.03	0.57	0.67
35%	25	0.63	0.12	0.02	0.58	0.68
40%	25	0.61	0.1	0.02	0.57	0.65
45%	25	0.63	0.1	0.02	0.59	0.67
50%	25	0.62	0.1	0.02	0.58	0.66

Table 4.7: The Shapiro-Wilk Results For Normality From 5% to 25% Treatment Levels.

	5%	**10%**	**15%**	**20%**	**25%**
Precision	Stat=0.91, p=0.03	Stat=0.51, p<0.05	Stat=0.61, p<0.05	Stat=0.57, p<0.05	Stat=0.85, p <0.05
Recall	Stat=0.87, p<0.05	Stat=0.78, p<0.05	Stat=0.79, p<0.05	Stat=0.72, p<0.05	Stat=0.69, p<0.05
F1-score	Stat=0.75, p<0.05	Stat=0.55, p<0.05	Stat=0.65, p<0.05	Stat=0.59, p<0.05	Stat=0.76, p<0.05

marizes the statistical comparison results, indicating no significant difference in Precision, Recall, and F1-score. Specifically, the results of the comparison for precision showed a test statistics of 7.96 and a p-value of 0.63. Likewise, a significant difference in the comparisons between the evaluation measures of Recall and F1-score (Recall Results: Test Statistics = 8.62, p-value = 0.56 , F1-score Results: Test Statistics = 8.56, p-value = 0.57) values were not found.

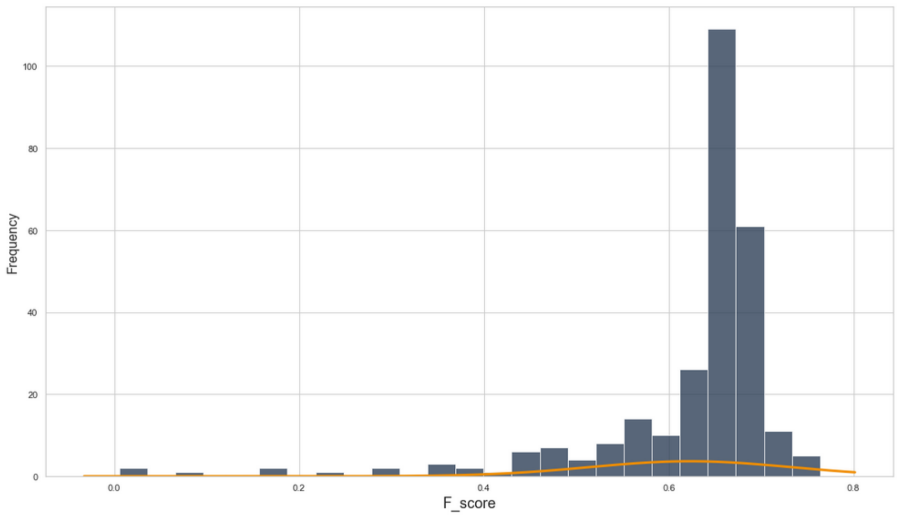


Figure 4.14: Frequency Plot For the F1-score Scores

Table 4.8: The Shapiro-Wilk Results For Normality From 30% to 50% Treatment Levels.

	30%	**35%**	**40%**	**45%**	**50%**
Precision	Stat=0.48, p<0.05	Stat=0.57, p<0.05	Stat=0.89, p=0.01	Stat=0.78, p<0.05	Stat=0.85, p<0.05
Recall	Stat=0.69, p<0.05	Stat=0.67, p<0.05	Stat=0.87, p<0.05	Stat=0.76, p<0.05	Stat=0.82, p<0.05
F1-score	Stat=0.55, p<0.05	Stat=0.6, p<0.05	Stat=0.89, p=0.01	Stat=0.74, p<0.05	Stat=0.78, p<0.05

Table 4.9: Statistical Results For the Comparison Between the Evaluation Measures Under All Treatment Levels.

	p-value	**statistics**
Precision	p=0.63	Stat=7.96
Recall	p=0.56	Stat=8.62
F1-score	p=0.57	Stat=8.56

Therefore, no statistical evidence could be found to support the rejection of the null hypotheses *H0p, H0r, H0f*.

4.7 Discussion

To answer the research question of *how to improve test case selection by handling class and attribute noise?*, we compare the results reported in Sections 4.6.1 and 4.6.2, and draw a comparison between the effectiveness of handling class noise and attribute noise. The comparison results are achieved by examining the precision, recall, and F1-score in Tables 4.4, 4.5, and 4.6, and Figure 4.8. Recall from Section 4.5.4.1 that the performance measures obtained at 0% treatment level (control group) are treated as the baseline measures. The remaining treatment levels are used to examine the effectiveness of handling attribute noise at different levels on the performance of the ML model used in MeBoTS.

By examining the performance measures in the Tables and Figure, the following observations are drawn from the comparison:

- compared with the other two trials of training, using an uncleaned data-set for training provides the lowest learning performance.
- training a learner on a class-noise cleaned data would improve the performance of the learner by 70% recall, 37% precision, and 59% F1-score, compared to a learner built on uncleaned data.
- training a learner on a class and attribute noise cleaned data results in almost no change in the prediction of passing test cases that are really passing (recall drop of 4%).

These observations imply that training a classifier on a class-noise cleaned data will yield to a better performance with respect to precision and recall than the other two Trials of training. Particularly, the results suggest that building a learner on class-noise cleaned data will allow testers to correctly exclude 8 out of 10 actually passing test cases from execution (81% precision). In addition, the results reveal that training a learner on a PANDA cleaned data would result in building a learner that is biased towards the positive class. The implication that these results bring in the domain of TCS are that tester would falsely exclude 5 out of every 10 actually passing test cases from execution. These results are in line with the conclusions drawn by Brodley and Friedl, and Zhu and Wu [21][54], which suggest that attribute noise is less harmful than class noise on the inductive performance.

Based on the results and discussion points, the following recommendations are suggested to testers:

- To avoid randomness in the prediction of test case verdicts, uncleaned data should not be used for building a learner for TCS.
- Testers should consider measuring the ratio of class noise in the data at hand before building a model for TCS. This would direct the testing effort by choosing an appropriate noise handling strategy. For example, if the ratio of class noise is small, then testers can rely on the robustness of ML algorithms without correcting or eliminating training instances. If the noise ratio is large, then testers would decide on a correction or elimination based strategy for cleaning noise.
- Testers should focus on cleaning class noise from the training data, but not necessarily the attribute noise.

4.8 Threats to Validity

When analyzing the threats to validity of our study, we followed the framework recommended by Wohlin et al. [46] and discuss the validity in terms of: external, internal, construct, and conclusion.

External Validity: External validity refers to the degree to which the results can be generalized to applied software engineering practices.

Test Cases Sample. Since our original uncleaned data are related to twelve test cases only, it is difficult to decide whether the studied sample of code churns is representative to the overall population. However, the selection of the studied sample was done randomly. This increases the likelihood of drawing a representative sample.

Control group. The control group used in this study consisted of a relatively small number of observations and attributes (19,815 observations and 800 attributes). This may pose a risk on the representativeness of the sample with respect to the overall population. However, the derivation of the control group was done by randomly selecting attributes and observations from the class-noise cleaned data. This increases the likelihood of drawing a representative sample in the control group.

Source code. In this study, we only used a single industrial program to examine the effect of class and attribute noise on the learning performance of a classifier. Therefore, we acknowledge that the generalization of the findings is difficult. However, since the goal of this paper is to gain an initial understanding of the effect of attribute and class noise, we accept this threat.

Nature of test failure. There is a probability of mis-labelling code changes in the original data if test failures were due to factors external to defects in the source code (e.g., machinery malfunctions or environment upgrades). To minimize this threat, we collected data for multiple test executions that belong to several test cases, thus minimizing the probability of identifying tests that are not representative.

Internal Validity Internal validity refers to the degree to which conclusions can be drawn about the causality between independent and dependent variables.

Configuration. In this study, the ranking of noisy observations produced by PANDA was determined using a bin size of five. Since the binning size in PANDA may affect the ranking of noisy observations [13], there is a likelihood that we chose a bin size that does not identify the highest noisy observations in the sample data. As a result, the applied treatment may not have eliminated all observations that come with the highest attribute noise. This may have an effect on the learning. However, our results showed that the standard deviations in the learning scores were not largely despaired across the 25 subjects, which means that the effect of the chosen bin size had a similar effect on learning across all experimental subjects.

Instrumentation. A potential internal threat is the presence of undetected issues in the scripts used for vector transformation, data-collection, and

PANDA's implementation. This threat was controlled by carrying out a careful inspection of the scripts and testing them on small subsets.

Machine Learning Model. The evaluation of learning was done using Random Forest only - the results were drawn from a single type of ML model. Hence, the tolerance of RF to noise and its performance will differ when using other types of learning algorithms. However, in this study, we focus on improving the learning performance by handling class and attribute noise irrespective of which model is most suited for noise tolerance.

Construct Validity Construct validity refers to the degree to which experimental variables accurately measure the concepts they purport to measure.

The Binning Algorithm. The binning algorithm used in the original work of PANDA was not explicitly stated in the original publication [13]. As a result, we used the sort_values function in the PANDA module of the scikit learn library to discretize attribute values into bins of predefined sizes. Thus, our implementation of the algorithm may differ than the one used in the original work. However, the authors of the original publication state that any binning algorithm can be used without affecting the performance.

The Calculation of Noise Score. The description for calculating the standardized noise score in the original publication of PANDA [13] created a confusion with respect to whether the mean and standard deviation should be calculated for each partition in x_j or x_k. On the one hand, the description states that *the standardized noise score for attribute value x_{ik} is calculated relative to the partitioned attribute value for instance i,$\hat{x}_{ik}$*. On the other hand, the description states that *'the mean and standard deviation of the non-partitioned attributes $x_{k,k\neq j}$ relative to each bin $\hat{x}_{j=0,\ldots,L'}$ is calculated'*. In our implementation, we interpreted the relativeness between an attribute value x_{i_k} with the partitioned attribute value for instance *i*, $\hat{x}_ik$ by subtracting the attribute value x_{ik} from the mean to standard deviation ratio of the bin in x_j relative to x_{ik}. The alternative interpretation would be to subtract x_{ik} from the mean to standard deviation ratio of the elements in x_k relative to the bin in x_j. Nevertheless, our implementation was manually inspected on a small set of line vectors (as shown in Section 4.3.4) and the ranking of noisy observations were in line with the definition of attribute noise provided in the original publication [13].

Majority class problem. Upon applying the treatment on the experimental subjects under the 10 levels, there is a chance that the prediction was biased towards one of the classes due to an imbalance in the distribution of classes. Due to the computational cost required to check the balance across 25 subjects for 10 treatment levels (250 trials), we could not validate that the post treatment subjects are balanced. Nevertheless, the results drawn from the learner's precision and recall (mean precision= 52, mean recall= 81) indicate that the learner was not biased towards a particular class.

Conclusion Validity Conclusion validity focuses on how sure we can be that the treatment we use really is related to the actual outcome we observe.

Differences among subjects. The descriptive statistics indicated that we have a few outliers in the sample. Therefore, we ran the analysis twice (with and without outliers) to examine if they had any impact on the results. Based on the analysis, we found that dropping the outliers had no effect on the results, thus we decided to keep them in the analysis.

4.9 Conclusion and Future Work

In this paper, we set off to study the effect of class and attribute noise in data on the learning performance of an ML model for test case selection. We chose to study the effect of handling the two noise types (class and attribute) using a correction and an elimination-based approaches. The results drawn suggest that handling class noise yields to a substantial improvement in the prediction of test case verdicts, whereas no similar conclusion could be drawn with respect to attribute noise. Our study provides empirical evidence which suggests that handling attribute noise is not necessarily important for building an effective learner for test case selection. This finding is counter-intuitive when considering the majority of related literature on attribute noise, which suggest that handling attribute noise improves the learning performance. This calls for more studies that need to examine the effect of handling attribute noise on learning in software engineering contexts.

There are still several questions that need to be addressed before concluding that handling class noise is more important than attribute noise. A first question is about finding whether other elimination approaches for identifying and handling attribute noise can have a different effect on learning than PANDA. A second question is whether similar results about the effect of class and attribute noise handling can be generalized when using other data-sets. Future research about the impact of class and attribute noise should experimentally explore the effect of both noise types by seeding class and attribute noise into a clean data-set and evaluating the learning effect. Other research directions include testing different approaches for handling class and attribute noise such as tolerance of different ML algorithms.

4.10 Appendix A

Attribute Noise	Performance metrics	Random Forest n_estimater=100	Random Forest n_estimater=300
0%	Acc	0.54	0.53
	Prec	0.53	0.50
	Rec	0.88	0.82
	F-score	0.66	0.62
	MCC	0.13	0.10
5%	Acc	0.54	0.53
	Prec	0.53	0.52
	Rec	0.83	0.84
	F-score	0.64	0.64
	MCC	0.1	0.10
10%	Acc	0.53	0.52
	Prec	0.51	0.51
	Rec	0.80	0.87
	F-score	0.61	0.64
	MCC	0.09	0.09
15%	Acc	0.53	0.52
	Prec	0.51	0.51
	Rec	0.78	0.93
	F-score	0.60	0.66
	MCC	0.08	0.10
20%	Acc	0.52	0.52
	Prec	0.50	0.51
	Rec	0.80	0.95
	F-score	0.61	0.66
	MCC	0.07	0.1
25%	Acc	0.53	0.52
	Prec	0.52	0.51
	Rec	0.88	0.95
	F-score	0.65	0.66
	MCC	0.10	0.07
30%	Acc	0.52	0.52
	Prec	0.50	0.51
	Rec	0.84	0.94
	F-score	0.62	0.66
	MCC	0.06	0.074
35%	Acc	0.53	0.53
	Prec	0.51	0.51
	Rec	0.85	0.91
	F-score	0.63	0.65
	MCC	0.1	0.11
40%	Acc	0.53	0.53
	Prec	0.53	0.51
	Rec	0.77	0.78
	F-score	0.61	0.61
	MCC	0.09	0.08
45%	Acc	0.54	0.53
	Prec	0.53	0.52
	Rec	0.82	0.85
	F-score	0.63	0.63
	MCC	0.13	0.10
50%	Acc	0.54	0.54
	Prec	0.53	0.52
	Rec	0.80	0.83
	F-score	0.62	0.64
	MCC	0.11	0.11

Chapter 5

Paper D

A Classification of Code Changes and Test Types Dependencies for Improving Machine Learning Based Test Selection

Al-Sabbagh, K.W., Staron, M., Hebig, R. and Gomes, F.

In Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering, pp. 40-49. 2021.

Abstract

Machine learning has been increasingly used to solve various software engineering tasks. One example of their usage is in regression testing, where a classifier is built using historical code commits to predict which test cases require execution. In this paper, we address the problem of how to link specific code commits to test types to improve the predictive performance of learning models in improving regression testing. We design a dependency taxonomy of the content of committed code and the type of a test case. The taxonomy focuses on two types of code commits: changing memory management and algorithm complexity. We reviewed the literature, surveyed experienced testers from three Swedish-based software companies, and conducted a workshop to develop the taxonomy. The derived taxonomy shows that memory management code should be tested with tests related to performance, load, soak, stress, volume, and capacity; the complexity changes should be tested with the same dedicated tests and maintainability tests. We conclude that this taxonomy can improve the effectiveness of building learning models for regression testing.

5.1 Introduction

Software testing has evolved to successfully accommodate for the growing demand of higher product quality and faster delivery of releases [112]. Nevertheless, testing has been notoriously costly for its massive resource consumption - accounting for more than 50% of the development life cycle. Therefore, optimizing testing processes becomes pivotal for companies of all sizes to reduce the cost overhead and increase the velocity of software development.

An essential yet costly activity in any testing process is to perform regression testing, which ensures that no new faults in the system arise due to making new changes to the code base. However, performing regression testing demands a large amount of resources and a long execution time, which makes it infeasible to run all impacted test cases on each committed code change.

To address this problem of regression testing, a number of test case selection approaches have been proposed in the literature [58], [78], and [113]. These approaches seek to improve the effectiveness of test case selection by inferring statistical models that can potentially predict affected test cases given changes in the code base. However, a mutual drawback among these approaches is that they omit to take into account the dependencies between specific types of code changes (e.g., memory and algorithmic changes) and test case types (e.g., performance and security tests) when training predictive models. For example, Al-Sabbagh et al. [78] proposed building a machine learning (ML) model for test selection by mapping history executions of test cases and their relevant code changes without considering what types of test cases are sensitive to the changes in the source code. Similarly, Knauss et al. [58] proposed an automatic recommender that analyzes the frequency in which test cases fail on a particular day given code changes made to software modules irrespective of the types of changes made in the code and their dependencies with specific test case types.

Therefore, in this paper, we set off to fill this gap by developing a facet-based taxonomy of dependencies between code changes and test cases of specific types. We define a *dependency* as a relation where a change in the source code of a given type that triggers a failure in one or more test cases of different types. The contribution of this work is two-fold. First, it gears the testing efforts at software companies by allowing the execution of test cases that are in relation with the submitted code changes to the development repositories - thereby potentially reduce the time for testing. Second, it lays down the foundation for researchers to investigate, expand, and refine the identified dependencies. The addressed research question is:

RQ: To which degree do software testers perceive content of a code commit and test case types as dependent?

To address this research question, we constructed a taxonomy, linking the test case types and the categories of source code that can trigger these test cases. First, we began the taxonomy building by identifying and extracting data from the literature to find the test types and categories of code changes and to identify potential synergies between them. Then, we surveyed testers from software companies to construct and design the faceted taxonomy [114] Finally, for two categories, where the survey results were inconclusive, we conducted a workshop with the testers to find the strength of these dependencies.

5.2 Related Work

Our work is related to studies on defect and testing taxonomies.

5.2.1 Defect Taxonomies

A widely applicable taxonomy in the software testing literature is Orthogonal Defect Classification (ODC), which was designed by Chillarege et al. [115]. The ODC taxonomy defines attributes for the classification of failures. Its main purpose was to identify the root cause of defects and to provide quick feedback to developers about defects' cause in the software process. The ODC can also be used for early detection of faults in static analysis. Several defect taxonomies have been built on the ODC as a starting point to develop different domain-specific taxonomies. For example, Li et al. [116] presented an extended taxonomy of ODC and named it Orthogonal Defect taxonomy for Black-box Defects (ODC-BD). The taxonomy was designed by the motive of increasing testing efficiency and improving the analysis of black-box defects. Evaluated on the analysis of 1860 black-box defects that belong to 40 software projects, the results showed that using ODC-BD reduced the testing effort by 15% in one month compared to the testing efficiency when not using the ODC-BD. Another work conducted by Li et al. [117] adopted ODC to classify web errors for an improved reliability. Their taxonomy classified web errors according to their response code, file type, referrer type, agent type, and observation time.

The primary focus of all related work described above is to improve the quality of the code base by identifying the root cause of defects and to gain insights into the types of commits that developers commit. However, our work aims to improve the testing process by providing a taxonomy of code changes and test cases that can be used to build classifiers for test case selection.

5.2.2 Taxonomies in Software Testing

Software testing has often been confronted with the challenge of unveiling software defects under sever time pressure and limited hardware resources. Due to its importance and practical relevance, several software testing taxonomies have been proposed in the literature. In a systematic literature review study [118], Britto identified a number of studies that present taxonomies in the area of software testing. The majority of these taxonomies, however, provides a classification of the suitability of testing techniques in different contexts. For example, Novak et al. [119] developed a tree-based classification of features that are attributed to existing static code analysis tools. The taxonomy offers a classification of existing static analyzers based on the technology, availability of rules, and the programming languages that each tool supports. Similarly, Vegas et al. [120] classified a set of unit testing techniques and mapped their characteristics with project characteristics to aid the selection of suitable testing approaches based on the project's characteristics. The presented taxonomy comprised a number of criteria such as when to use the testing approach, who to use it, and where it can be used. Felderer et al. [121] presented a classification for supporting the categorization of risk-based testing approaches and tailoring their usages depending on the context and purpose. The taxonomy classifies different risk drivers, risk assessments, risk-based test processes. All of these taxonomies provide a generic classification of the applicability of testing techniques in different software engineering projects. However, no taxonomy discusses the dimension of dependencies between code commits and test case types. Classifying these dependencies can potentially aid in the identification and execution of tests that are relevant to the committed code and hence counteract exhaustive testing efforts. The taxonomy presented in this study aims at filling this gap by identifying facets of dependency connections from the viewpoints of software testers.

5.3 Research Method

In this study, we follow the method proposed by Usman et al. [48] to guide the construction of the taxonomy. The method comprises of four phases: i) planning, ii) identification and extraction, iii) design and construction, and iv) validation.

5.3.1 Planning

The first phase in the adopted method involves six activities for planning the context of the taxonomy and defining its initial settings. Table 5.1 illustrates the outcome of each planning activity. Since the ultimate goal of this study is to gear the testing efforts by improving the selection of test cases, then the the knowledge area associated to the taxonomy is in the domain of software testing (A1). The second activity (A2) defines the objective of the taxonomy, which in our case is to identify the degree at which testers perceive dependency patterns between code changes and test case types. The subject matters (units of classifications) are categories of code changes and test case types (A3). A faceted-based approach is devised for creating the taxonomy (A4). The procedure for classifying the subject matters are qualitative and quantitative - literature review, survey, and discussions with testers in a workshop setting (A5). Finally, the basis of the taxonomy consists of categories of code changes and test case types drawn from the literature (A6).

5.3.2 Identification and Extraction

The identification and extraction phase involves identifying the main categories and terms used in the taxonomy. We begin the implementation of this phase by

Table 5.1: Planning Activities

Id	Planning Activity
A1	The software engineering knowledge associated to the designed taxonomy is software testing.
A2	The main objective of the proposed taxonomy is to identify dependency patterns between code changes and test case types from the perspective of testers.
A3	The subject matters of the designed taxonomy are categories of code changes and test case types.
A4	The taxonomy was designed using a facet-based structure.
A5	The procedure used for classifying the subject matters was qualitative and quantitative.
A6	The basis of the taxonomy consists of code change categories and test case types drawn from the literature.

reviewing the literature in search for knowledge about the subject matters. For this purpose, we account for two inclusion criteria in our literature search. First, we wanted to include papers that discuss the impact of specific changes in the code on the quality of the system. Second, we were only interested in papers that were written in English and accessible. The challenge in this phase was to extract terms that are consistent and not interchangeably used in different research studies. Therefore, to overcome this challenge we based our literature search on the set of recognized test case types defined in the international standard ISO/IEC/IEEE CD 2911901:2020(E) document [55] (presented in Section 5.4.1). That is, for each test case type in the ISO document, we searched for relevant papers that empirically investigate or theoretically discuss types of code changes that trigger a reaction among the test cases. The outcome of this phase was a list of six categories of code changes and 18 test case types. Further, and based on our literature search, we identified synergy links between the six code categories and the 18 test types (as depicted in Fig 5.2).

5.3.3 Design and Construction

This phase presents the relationships between the identified categories and describes how they were connected. Since the goal of the taxonomy is to answer the question of *To which degree do software testers perceive content of a code commit and a test case types as dependent?*, we decided to open up for the community of testers to seek their opinions about potential dependency patterns between the categories of code changes and test case types and to identify the strengths of the identified dependencies.

5.3.3.1 Survey

We began this phase by creating a survey and distributing an invitation email to software development companies that are affiliated to a Swedish consortium called 'Software Center'. The consortium comprises a total of fifteen companies and five universities that collaborate together to advance knowledge in seven different software engineering themes.

To mitigate the risk of receiving responses from different domain perspectives (e.g., web development), we decided to focus on surveying testers that specialize in the same domain area. Therefore, we sent the invitation email to five companies that are active in the development of embedded systems. The survey comprised two column lists. The first list included definitions of the test case types (see Section 5.4.1), whereas the second list included the categories of code changes (see Section 5.4.2). As a first task, all invitees were asked to provide a mapping between each test case type and category of code changes, where a mapping corresponds to a dependency between a single test case type and a category of code change.

The second task was for testers to propose and map additional test case types with categories of code changes that were not provided in the survey. The purpose was to mitigate the risk of missing out dependency patterns that testers perceive as important.

Finally, to achieve a better understanding of our target group of testers, all invitees were asked to mark the test case types that they exercise in their workplaces. Overall, we received a total of nine responses from nine testers working at the three software development companies. A general overview of the number of experienced testers for each test case type is provided in Fig 5.1.

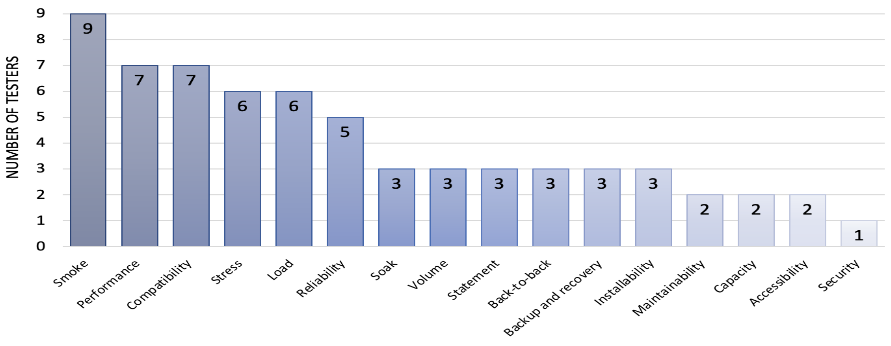


Figure 5.1: Number of Experienced Testers Per Each Test Type.

5.3.3.2 Workshop with Testers

The data from the survey provided us with the understanding of the dependencies. However, these dependencies could be of different strength and therefore we organized a workshop with the respondents to assess the strengths of dependencies for each test type to code changes. Three out of the nine respondents, who participated in the survey, and three other testers from another software company attended the workshop. Our analysis of the survey responses showed that the strongest dependencies were concentrated around the memory management and complexity categories of code changes. Therefore, we decided to focus on assessing the dependency strengths between these two categories of code changes and test case types in the workshop.

During the workshop, the entire group of testers discussed how sensitive each test type to the change of source code that affects 1) memory management or 2) complexity. The goal of the discussion was to gain an understanding of the dependency strengths from the viewpoint of testers, in the following scale:

1. Not sensitive at all. This level was used when the testers judged that such a change would not trigger the test case to fail.

2. Not very sensitive. This level was used when the testers judged that triggering a failure would be coincidental.

3. Somewhat sensitive. This level was used when the testers judged that triggering would be under specific conditions.

4. Sensitive. This level was used by the testers to indicate that a change under most conditions triggers a test case failure.

5. Very sensitive. This level was used when the change should trigger the failure of the test case.

After discussing the sensitivity strengths, using the above scale, we asked the testers to justify their views about the sensitivity of each dependency by providing explanations for their ranking.

5.3.4 Validation

This phase ensures that the selected subject matters are clear and thoroughly classified [48]. This can be achieved using three distinct methods: Orthogonality demonstration, benchmarking and utility demonstration. Most of the taxonomies proposed in Software Engineering are evaluated via an utility demonstration, i.e., authors apply their taxonomy to an example [48]. In turn, benchmarking is used to compare the classification capabilities of different taxonomies. In both cases, the taxonomy needs to be applied in actual software artefacts. For this study, we cannot perform those types of validation because we do not have access to test cases or code changes from our industry partners. Therefore, we validate our taxonomy using an orthogonality demonstration. That is, we demonstrate and discuss the orthogonality between strongly dependent categories from the viewpoints of testers. The goal is to illustrate the unique classifications offered by our taxonomy. Based on this demonstration, we aim to highlight which types of tests map to unique types of code changes, as well as those dependencies that cover multiple types of tests.

5.4 Results

This section presents the findings for the research question *To which degree do software testers perceive content of a code commit and a test case types as dependent?*

5.4.1 Test Case Types

In this paper, we decided to base our literature search for extracting code change categories on the list of test case types defined in this ISO/IEC/IEEE CD 2911901:2020(E) document [55]. This was done to overcome the challenge of encountering different terms of test case types that are used interchangeably in published articles. For example, the terms ‘back to back’and ‘differential’testing can be found and used interchangeably in the literature. Table 5.2 lists the definitions of all test case types that we used in our literature search. We used each test case type in the Table to search for relevant papers that empirically investigate or theoretically discuss the dependency between the relevant test case type and code changes.

5.4.2 Code Change Categories and Dependencies with Test Case Types

Our literature search returned a set of 16 relevant papers from which we could extract six different categories of code changes. These categories were: 1) Memory Management, 2) Complexity, 3) Design, 4) Dependency, 5) Conditional, 6) Data. Based on the literature search, we identified 21 dependency links between the six drawn categories of code and eight out of the 18 test case types defined in the ISO document, as shown in Fig 6.1. Each dependency corresponds to a relation where a change in one of the code category results in a failure of a test case of specific type.

We now define the identified categories of code changes and illustrate the effect of each on test case types by means of code examples written in the C++ language.

Memory management: This category of change involves groups that are concerned with the management of memory occupied by the system during run-time. Such changes include introducing/fixing memory leaks, buffer overflow, dangling pointers, and resource interference with multi-threading. The following test types would react to this category of change: performance [122], load [123], security [124][125], soak [126], stress [127], reliability [128] tests. A common memory leak scenario occurs when a developer allocates memory space using the *new* or *malloc* keywords, and misses freeing memory space after they were used. As the program grows in size, less memory becomes available and thereby a performance degradation is encountered. The code example in Fig 5.3 shows

Table 5.2: Definitions of Test Case Types

Test Type	**Definition**
Smoke	Initial testing of the main functionality of a test item to determine whether subsequent testing is worthwhile.
Soak	Testing performed over extended periods to check the effect on the test item of operating for such long periods.
Stress	Testing performed to evaluate a test item's behaviour under conditions of loading above anticipated requirements.
Volume	Testing performed to evaluate the capability of the test item to process specified volumes of data in terms of capacity.
Load	Testing performed to evaluate the behaviour of a test item under anticipated conditions of varying loads.
Statement	Test design technique in which test cases are constructed to force execution of individual statements in a test item.
Maintainability	Evaluate the degree of effectiveness and efficiency with which a test item may be modified.
Security	Evaluate the degree to which a test item, and associated data, are protected against unauthorized access.
Performance	Evaluate the degree to which a test item accomplishes its designated functions within given time.
Capacity	Evaluate the level at which increasing load affects a test item's ability to sustain required performance.
Portability	Evaluate the ease with which a test item can be transferred from one environment to another.
Installability	Testing conducted to evaluate whether a set of test items can be installed as required in all specified environments.
Compatibility	Measure the degree to which a test item can function along-side other independent products.
Reliability	Evaluate the ability of a test item to perform its required functions under stated conditions for a period of time.
Accessibility	Determine the ease by which users with disabilities can use a test item.
Back-to-back	An alternative version of the system is used as an oracle to generate expected results for comparison from the same inputs.
Backup and recovery	Measures the degree to which a system state can be restored from backup within specified time in the event of failure.
Procedure	Evaluate whether procedural instructions for interacting with a test item to meet user requirements.

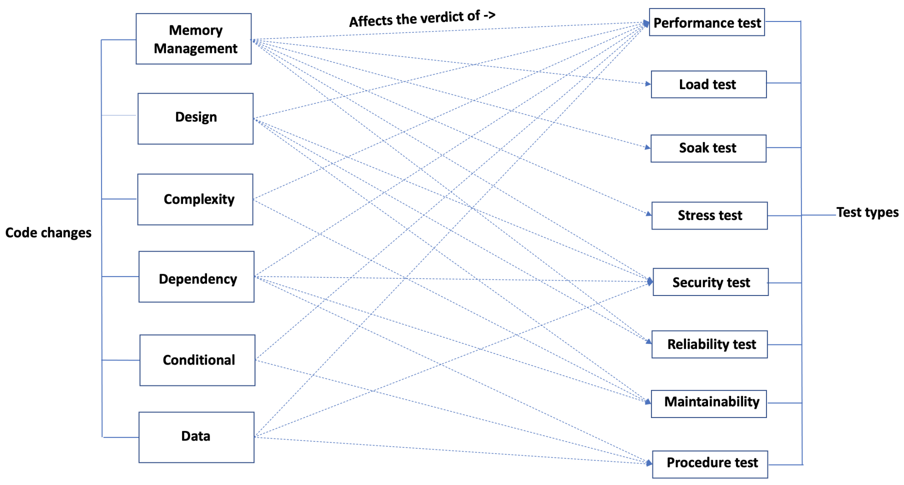
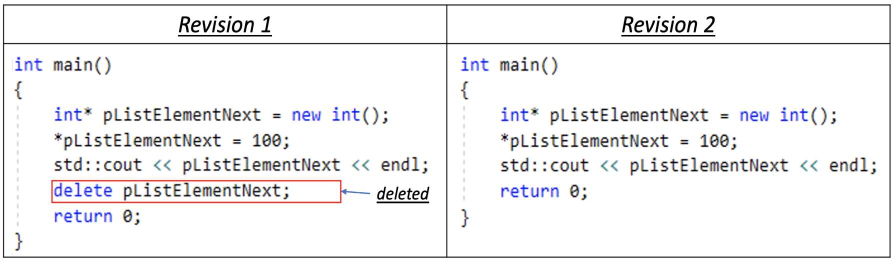


Figure 5.2: Extracted Categories of Code Changes and Their Dependency with Test Case Types.



```
int main()
{
    int* pListElementNext = new int();
    *pListElementNext = 100;
    std::cout << pListElementNext << endl;
    delete pListElementNext;
    return 0;
}
```

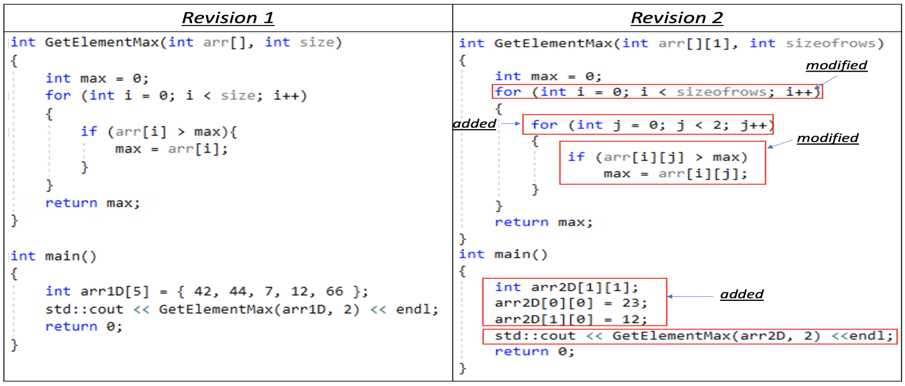
```
int main()
{
    int* pListElementNext = new int();
    *pListElementNext = 100;
    std::cout << pListElementNext << endl;
    return 0;
}
```

Figure 5.3: Code Example For Memory Management Change.

how the memory space allocated for pointer `pListElementNext` was unfreed from the memory after being used in revision 2.

Complexity: This category represents changes that add/reduce the time complexity of the program. It includes changes such as adding or removing loops, conditional statements, nesting blocks and/or recursions. The following test types have been identified to react to this category of change: performance [129], [130], maintainability [131], [132] tests. Fig 5.4 shows a code example for finding the maximum integer element in an array. The function in the first revision takes a one dimensional array as input, whereas the second revision is modified to accept two-dimensional arrays. The nested loop added to the function in revision 2 would result in an increased time complexity order. Similar changes can potentially trigger performance degradation and thereby performance test failures.

Design: This category involves changes that include code refactoring, adding or removing methods, classes, interfaces, and enumerators, and code smells. The following test types have been identified to react to this category of change:



Revision 1:

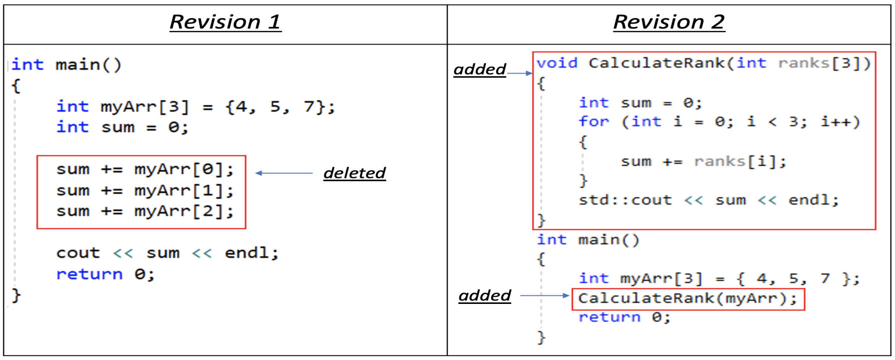
```
int GetElementMax(int arr[], int size)
{
    int max = 0;
    for (int i = 0; i < size; i++)
    {
        if (arr[i] > max){
            max = arr[i];
        }
    }
    return max;
}

int main()
{
    int arr1D[5] = { 42, 44, 7, 12, 66 };
    std::cout << GetElementMax(arr1D, 2) << endl;
    return 0;
}
```

Revision 2:

```
int GetElementMax(int arr[][1], int sizeofrows)
{
    int max = 0;
    for (int i = 0; i < sizeofrows; i++)
    {
        for (int j = 0; j < 2; j++)
        {
            if (arr[i][j] > max)
                max = arr[i][j];
        }
    }
    return max;
}
int main()
{
    int arr2D[1][1];
    arr2D[0][0] = 23;
    arr2D[1][0] = 12;
    std::cout << GetElementMax(arr2D, 2) <<endl;
    return 0;
}
```

Figure 5.4: Code Example For Complexity Change.

maintainability [131], performance [131], security [133], and reliability [134]. The code example in in Fig 5.5 illustrates a design change in a program that computes the sum of an array elements. The function 'CalculateRank' was added in the modified revision to handle the task of summing up the array elements. Such design decisions reduce the amount of code lines in the program and thus improves its maintainability.



Revision 1:

```
int main()
{
    int myArr[3] = {4, 5, 7};
    int sum = 0;

    sum += myArr[0];
    sum += myArr[1];
    sum += myArr[2];

    cout << sum << endl;
    return 0;
}
```

Revision 2:

```
void CalculateRank(int ranks[3])
{
    int sum = 0;
    for (int i = 0; i < 3; i++)
    {
        sum += ranks[i];
    }
    std::cout << sum << endl;
}
int main()
{
    int myArr[3] = { 4, 5, 7 };
    CalculateRank(myArr);
    return 0;
}
```

Figure 5.5: A Code Example For Design Change.

Dependency: This category describes a code change that involves adding/ removing/ modifying a dependency to another module/ library. It can be importing/ removing/ modifying a new library, a new namespace, or a new class. Changes in the dependencies between software artefacts can trigger the following tests: maintainability [135], security [124], procedure [136], and performance [130].
Conditional: This category of change occurs when a logical operator or a comparative value in a condition is modified. A misuse of logical expressions might result in generating the wrong outputs. Performance and procedure tests [130][136] were identified as dependent on this category of change.
Data change: This category involves 1) changing functions' parameters, 2)

passing parameters of incompatible types to modules/ functions, and 3) adding/ fixing assignments of incompatible types to variables, casting statements, and array size allocations, and 4) modifying variable declarations. The following tests would react to such code changes: security [137], performance [130], and procedure [136].

5.4.3 Dependency Patterns and Strengths

5.4.3.1 Survey.

Based on the types of tests and code changes extracted in the previous step, we created the survey. We sent our survey to 15 industry practitioners and received responses from nine participating testers (i.e., 60% response rate). Our analysis focuses on 1) examining whether testers had proposed additional types of test cases or categories of code change, and 2) examining the level of agreement and disagreement between the testers' perceived connections of types of tests and code changes. For instance, whether testers expect a connection between design changes and maintainability tests, as reported in the literature. Fig 5.6 is a contingency table that depicts the testers' opinions about potential dependencies. Our analysis of the responses revealed the following observations:

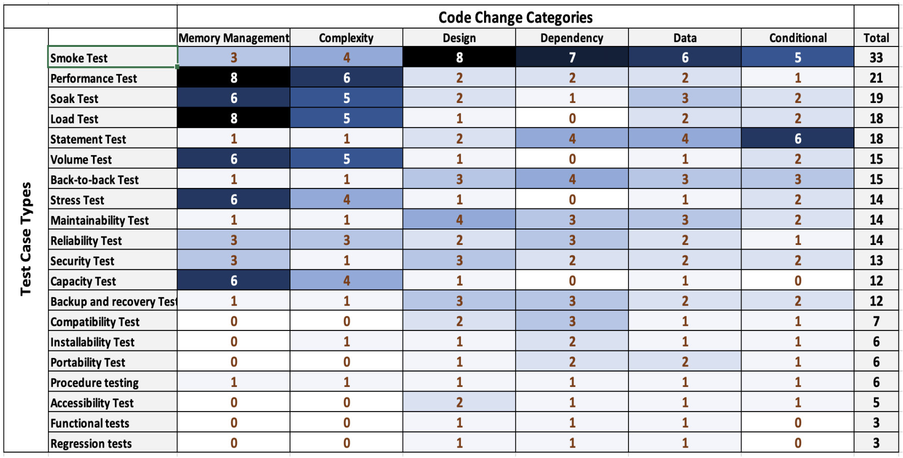
- The strongest dependency patterns were mostly concentrated around the memory management and complexity categories of code changes.

- There was a general consensus between the testers about the mappings between performance, soak, load, stress, capacity, and volume tests and the six types of code change categories.

- Most of the discrepancies in the responses were in the classification of the design, dependency, and data categories.

- Two additional test types, i.e., not found in our literature extraction, were proposed by the testers: Regression and functional tests. The ISO/IEC/IEEE CD 2911901:2020(E) considers these two types of tests as testing activities, since these can be applied at any point in time irrespective of the testing level (unit, integration, system, and user acceptance) [55].

Due to the agreement between most testers about the connection between the complexity and memory management categories of code changes, we decided to focus the workshop on exploring the deeper connections between these two types of code changes and all types of tests. Focusing on only those two categories allowed us to capture the details of practitioners' perception about the connections between code changes and many types of tests such as process or human factors related to identifying those changes, or code constructs used in industry to classify those changes.



		Code Change Categories						
		Memory Management	Complexity	Design	Dependency	Data	Conditional	Total
Test Case Types	Smoke Test	3	4	8	7	6	5	33
	Performance Test	8	6	2	2	2	1	21
	Soak Test	6	5	2	1	3	2	19
	Load Test	8	5	1	0	2	2	18
	Statement Test	1	1	2	4	4	6	18
	Volume Test	6	5	1	0	1	2	15
	Back-to-back Test	1	1	3	4	3	3	15
	Stress Test	6	4	1	0	1	2	14
	Maintainability Test	1	1	4	3	3	2	14
	Reliability Test	3	3	2	3	2	1	14
	Security Test	3	1	3	2	2	2	13
	Capacity Test	6	4	1	0	1	0	12
	Backup and recovery Tes	1	1	3	3	2	2	12
	Compatibility Test	0	0	2	3	1	1	7
	Installability Test	0	1	1	2	1	1	6
	Portability Test	0	0	1	2	2	1	6
	Procedure testing	1	1	1	1	1	1	6
	Accessibility Test	0	0	2	1	1	1	5
	Functional tests	0	0	1	1	1	0	3
	Regression tests	0	0	1	1	1	0	3

Figure 5.6: Testers' classifications of code changes and test case types. Each cell indicates the number of testers that perceive a relationship between the corresponding type of code changes and tests. Darker cells indicate stronger level of agreement between testers.

5.4.3.2 Workshop

We now present the results of the dependency scores given by the testers during the workshop. Figs 5.7 and 5.8 are diverging plots that show the sensitivity strengths of each test type to the memory management and complexity categories. By examining the sensitivity strength scores, of each test case type in Fig 5.7, we observe that the majority of the testers perceived six tests types to be mostly sensitive to memory management changes. Namely, performance, load, soak, stress, volume and capacity tests. Similarly, Fig 5.8 shows that performance, soak, load, statement, stress, volume, and maintainability tests were perceived as mostly sensitive to complexity related changes. In the remainder of this subsection, we present the main results of the discussions with the testers that explain their perspective on those connections.

5.4.3.3 Memory Management

Smoke, back-to-back, and statement tests: The respondents justified the low sensitivity strengths of these three test types to the fact that they focus on the functionality of the software system, rather than its qualities. One respondent linked the sensitivity of smoke tests to memory management changes to two specific scenarios: 1) when changing from one programming language to another, or 2) when doing major code refactoring.

> *"It's not that often that the smoke tests will break due to memory management changes but one possible scenario for this to happen is when we switch from C to C++ first we changed the compiler, then we started modernizing the code to use smart pointers. Another scenario is when we do major refactoring to optimize the code base."* — Participant 1

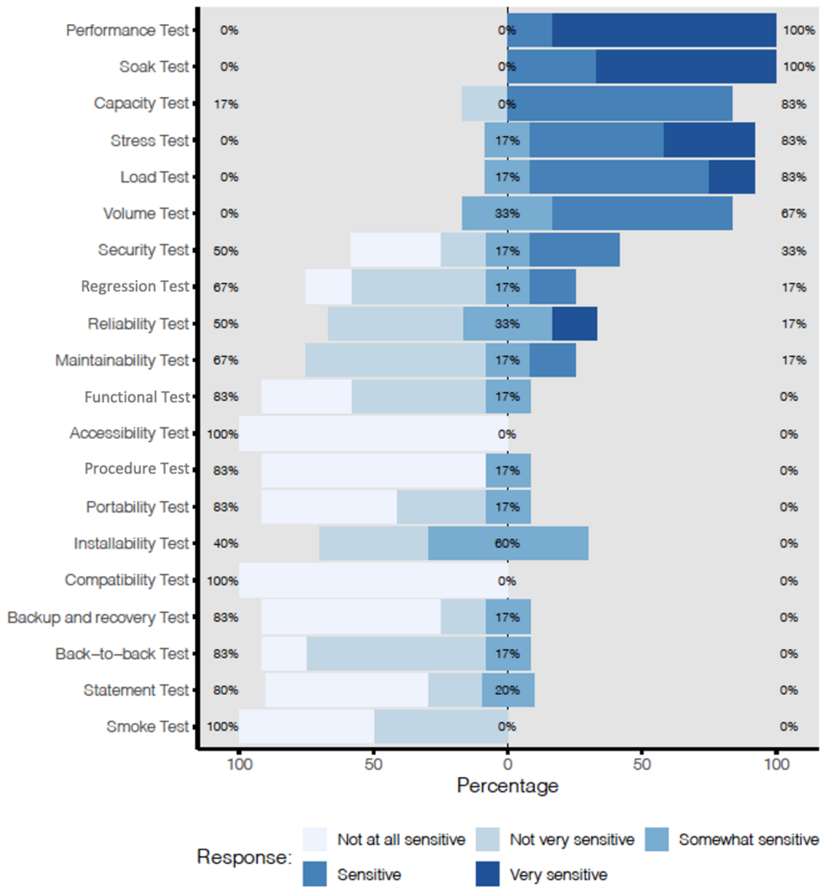


Figure 5.7: Diverging plot showing the strength of perceived connections between each test type and memory management changes. The percentages to the right indicate the proportion of testers that see a stronger relationship, in contrast to those that see a weaker relationship. Testers with a neutral view are shown as the percentage in the middle.

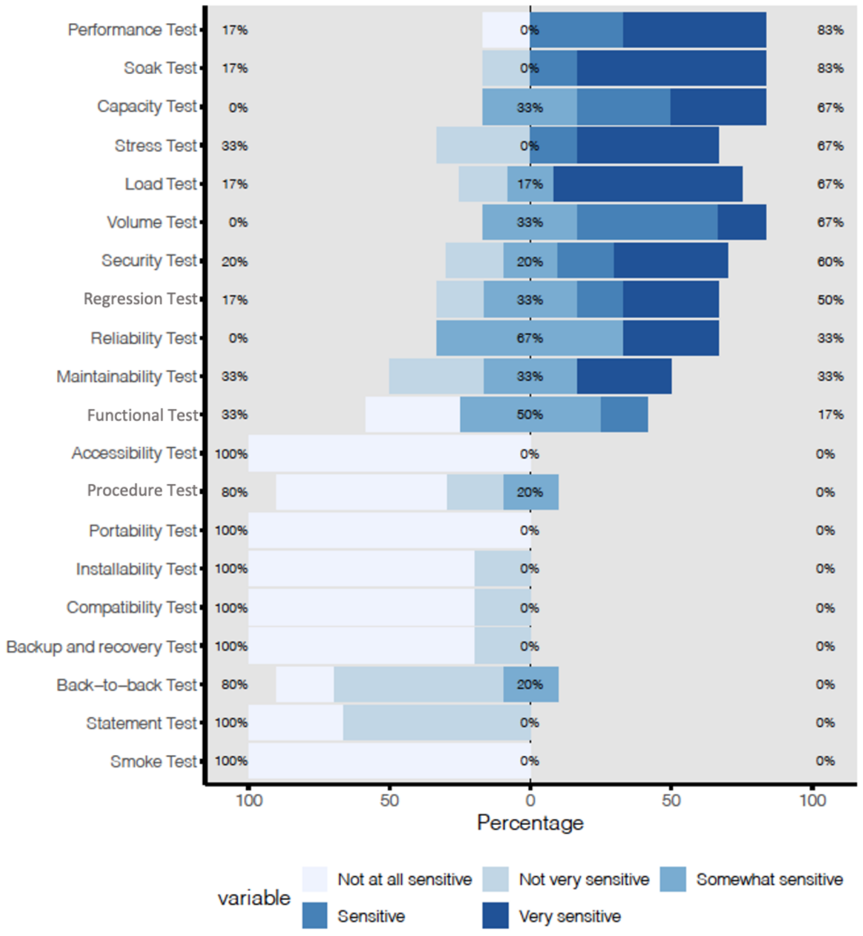


Figure 5.8: Diverging plot showing the strength of perceived connections between each test type and complexity changes. The percentages to the right indicate the proportion of testers that see a stronger relationship, in contrast to those that see a weaker relationship. Testers with a neutral view are shown as the percentage in the middle.

Compatibility and portability tests: All testers agreed that these two types of tests are not sensitive at all to memory changes. The testers explained that these tests may only be triggered in the event of hardware failure in the environment. One opposing viewpoint considered memory management changes to have an effect on the stability of APIs used for information exchange in a shared environment, and thereby can trigger a failure in the two tests.

> *"Failure in these two types of tests can be explained by a device failure or in the way the APIs in the shared environments are handling concurrent requests, which often requires memory management changes."* — Participant 1

Load, stress, soak, capacity, and volume tests: The majority of testers considered these test types to be very similar to performance tests. As a result, most of the justifications given about the sensitivity strengths of the five tests are somewhat similar. The testers explained that, in general, failure in one of the five test types can be triggered by memory related changes when expanding the functionality of existing classes.

> *"if you allocate more memory to expand an existing class then failure among performance tests might be triggered."* — Participant 2

In addition, one tester emphasized that failure in any of these tests depends on the amount of changes made between releases and the information specified in the test oracle. That is, failures can only be captured when the amount of code changes made between releases is large.

> *"Failure in these tests depends on the oracle. If you just use the performance test to compare performance from the latest release then there might be no issues because the changes are too small, but if you do big changes then you might spot memory problems."* — Participant 2

Installability tests: The sensitivity of this test type was perceived as moderate (somewhat sensitive) by 50% of the testers. These testers argued that installability testing is sensitive to memory management changes in situations where the development team decides to change from one operating system to another.

> *"When porting from a Windows environment to a Linux environment, we should make some memory changes, which trigger installability tests to fail."*
> — Participant 3

Security tests: There was a disparity in the views of testers regarding the sensitivity of this test type. 33% of the testers perceived this test to be sensitive to memory changes, 17% perceived it to be somewhat sensitive, whereas 50% of testers perceive a low sensitivity to this type of test. Testers who considered this test type to be sensitive argued that memory changes lead to memory leaks which, if not properly managed, might expose the system to security breaches.

> *"I think that memory management changes could lead to things being exposed that should not be. For example exposing kernels space memory to be violated."*
> — Participant 1

Disagreeing participants argued that resource leaks result in performance issues rather than security breeches. Further, they linked the sensitivity of security tests to the program domain.

> *"In specific domains, memory management is mostly handled on the cloud side providing the service. Internally, memory is not something that will trigger security tests to fail."* — Participant 4

5.4.3.4 Complexity code changes

Performance, soak, load, volume, and stress tests: The majority of the testers ranked these types of tests to be either sensitive or very sensitive to complexity changes. As an argument for their ranking, the testers discussed that adding complexity changes such as nested loops will increase the cyclomatic complexity size in the system, which would in turn affects the system's response time.

> *"As the cyclomatic complexity increases, the response time of the system will also get impacted."* — Participant 2

The remaining minority of the testers argued that developers are aware of the impact of adding complexity changes on performance. As such, it is highly unlikely that developers will commit complexity code changes without optimizing their code before testing.

> *"If developers are adding complexity consciously then there will be performance issues, but often the times, developers will address these complexity before even pushing their code for testing."* — Participant 3

Maintainability test: All of the participants perceived this test type to be either sensitive or very sensitive to complexity changes in the code. One of the participants argued that adding more control paths in the system, such as loops and case blocks, leads to the development of larger and poorly structured software, which makes it more difficult and less efficient to maintain.

> *"Adding things like loops or method calls into the program increases its size and makes the task of debugging more difficult as the program evolves over time."* — Participant 5

Security test: 50% of the participants indicated that security tests are somewhat sensitive to complexity changes. This was explained by the fact that adding recursion calls and loops to the code can potentially increase the size and modularity of the system under test, thus it will increase risk of missing security vulnerabilities. Conversely, around 30% of the participants believed that security tests are not sensitive at all to complexity changes. This contrasting view indicates that the links between security threats and increasing/decreasing code complexity are not clear for testers.

> *"I think it's not really a good thing to add complexity for security aware purposes. It is very important to understand what's going on in the code to be able to deal with things like security."* — Participant 2

> *"adding loops will in no way expose the system to external threats and therefore no security tests will break if more loops are added - adding loops will not cause any vulnerabilities in the system."* — Participant 6

The remaining 20% of the participants considered security tests to be sensitive to complexity changes, but did not provide any justification for this rank.

5.4.4 Resulting Taxonomy

The constructed taxonomy is based on the analysis of the overall agreement between testers who participated in the workshop and their justifications about each dependency. A test case type whose overall sensitivity to a code change was ranked as either sensitive or very sensitive by the majority of the testers was added to the taxonomy - provided that a justification for the dependency was made by one or more of the agreeing testers. Our analysis results of the workshop discussions show that testers have an aligned viewpoint with the classifications drawn from the literature in six of the dependency connections. Namely between: 1) memory management code and performance, load, soak, and stress tests, 2) complexity code and performance and maintainability tests. Beside these aligned dependencies, testers perceive six other dependencies to be in a strong causality relationship with the two categories of code. Those dependencies were between 1) memory management code changes and volume and capacity tests, 2) complexity code changes and load, soak, stress, and volume tests. Fig 5.9 shows the constructed taxonomy. We identify the strong and weak relationships mentioned by practitioners. Overall, the results show that the memory management code should be tested with tests related to performance, load, soak, stress, volume and capacity; the complexity changes should be tested with the same and additionally with the dedicated maintainability tests.

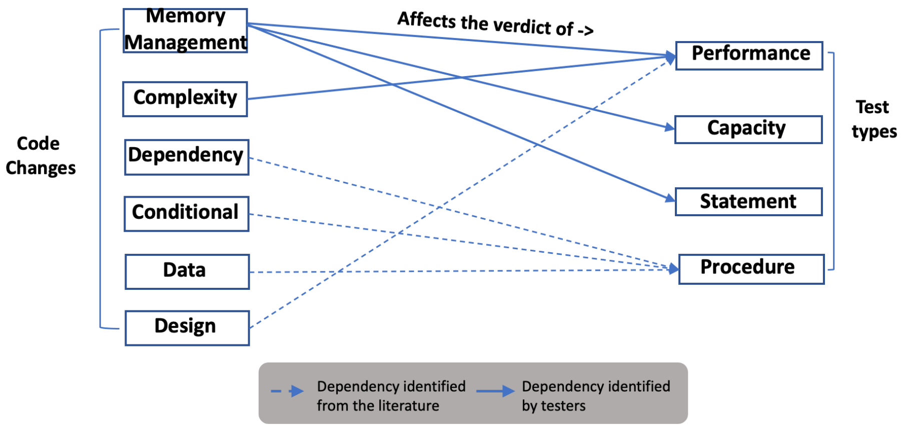


Figure 5.9: The final taxonomy of code changes and test case types. The solid connectors represent strong dependencies perceived by practitioners, whereas the dashed connectors correspond to those dependencies perceived as weak.

5.5 Taxonomy Validation

We evaluate our taxonomy by discussing the orthogonality of its classification. In other words, we illustrate how the chosen facets can support the prediction of connections between types of tests and code changes. Particularly, we emphasize the unique combinations found in our facets for supporting testers to classify the tests in connection with the code changes made. We frame the applicability of our taxonomy in relation to automated prediction of relationships between code and tests to support effective test orchestration.

5.5.1 Orthogonality of the Taxonomy's Facets

The majority of relationships are connected to the memory management code changes (11/18). That is not surprising as most of the types of tests found in literature cover system qualities. In fact, during workshops, practitioners rarely mention updates in functionalities (e.g., system requirements), except when discussing complexity changes. Memory management is exclusively connected with 5 test types, such that only 1 of those connections is strong (capacity tests). Consequently, those weak connections can be used to avoid overhead in test executions when focusing the verification of changes in memory management of software systems. Changes in complexity have fewer connections and most of them are actually shared with memory management (6/7), hence indicating a confounding factor between verifying changes in complexity to their impact on verifying memory management. Maintainability is only associated with complexity which is not surprising, since the complexity of a source code has impact on core aspects of maintainability such as testability and debugging [138]. The results show one weak connection shared between both types of code changes, which is related to security testing. Still, practitioners did not seem to have a consensus on how to handle security tests. Note that on Figs 5.3 and 5.4, security is ranked in the middle between the more explicit agreement and disagreements for both code categories. These contrasting views from practitioners on the purpose of security tests align with the findings drawn by Morrison et al. [139], where the authors highlighted a number of factors that impede the construction of effective vulnerability ML models.

5.5.2 Instrumenting Prediction of Dependencies

Table 5.3 breaks down memory management and complexity changes into specific types and their connection to specific code constructs. We choose C++ constructs because our study encompasses the domain of embedded systems. Future work aims at expanding the constructs to other programming languages such as Java or Python. Associating these code changes to specific code constructs enables automatic extraction and identification of code changes by using information from control version systems, such as git. The process of identifying and classifying code lines into their relevant categories can be

Table 5.3: Types and Constructs Related to Memory Management and Complexity Code Changes.

Memory Management		
Subcategory	**Description**	**Code Constructs**
Dangling/ Wild pointers	occurs when deleting an object from memory without altering the pointer that points to the object's location.	&variable, *variable, NULL, free
Memory leaks	occur when memory space is allocated but not freed. If such incidents occur, leaks will happen and could eventually cause the program to run out of memory resulting in a program halt.	delete, free, new, malloc
Buffer overflow	occurs when the data gets written past the boundaries of the buffer allocated in memory.	malloc, strcpy, gets, strcmp

Complexity		
Subcategory	**Description**	**Code Constructs**
Loops and conditions	repeating a sequence of instructions for n times until one or more conditions are satisfied. The repetition can occur in the form of multiple nested loops.	for, while, do, if, switch, case, break
recursion	Occurs when a function calls itself until an exit condition is satisfied.	

instrumented using, for example, a tokenizer and a lexicon of vocabulary that contains a mapping between code tokens (constructs) and their relevant categories of code. For example, a code line that appears with a combination of the tokens *'delete, free, new, and malloc'* can be used to classify a code line as memory management related, since these tokens are used during objects' creation/destruction (Table 5.3). In contrast, automatically identifying and extracting types of tests is more challenging because those tests are used across different levels (e.g., unit or system) such that keyword extraction is inaccurate, particularly for higher levels of testing where tests are written in natural language (e.g., acceptance tests). Therefore, for this study, we assume that practitioners have access to the types of their tests, as part of their test process.

RQ. To which degree do software testers perceive content of a code commit and test case types as dependent?

The measured degree of perception among software testers suggests a strong dependency between performance, load, soak, stress, and volume tests and memory management related code changes. On the other hand, testers believe that soak, statement, back to back, security and installability tests are in weak dependencies with memory management code. Similarly, the majority of testers perceive the same set of strongly dependent test types with memory management changes to be dependent on complexity changes; in addition to maintainability tests and excluding capacity tests.

Based on these findings, test orchestrators that are keen on using ML models for test selection are encouraged to build their ML models on data that reflects the dependency patterns depicted in the presented taxonomy (Fig 5.9). Particularly, by mapping memory management and algorithmic complexity related code changes to the verdict of the strongly dependent test case types.

5.6 Threats to validity

In this section, we briefly discuss the limitations of our paper using the framework recommended by Wohlin et al. [46].

Conclusion Validity: Since this paper does not aim to provide a systemic survey, we did not use a formal protocol for conducting the literature review. Therefore, we cannot ensure that the selection of the code categories and test case types was unbiased. However, we minimize this risk by inviting testers to propose other types of code changes and test cases that are not provided in the survey invitation email. Moreover, there is a likelihood that we missed adding valid dependencies in the taxonomy as a result of 1) not discussing the sensitivity of all test types with testers, and 2) lack of experience among testers in some test case types. However, since the goal of this work is to study the dependency between code changes and test types, we accept this risk.

External Validity: The sample size of testers who participated in the survey and the workshop was small. Therefore, we acknowledge that the generalization of our findings might be delimited. However, the survey data and the workshop discussion provided some valuable insights into understanding the dependencies and sensitivity strengths of different test case types and code changes.

Internal Validity: The time span between the distribution of the survey and the the workshop was almost two months. This poses a threat with respect to the testers' comprehension of the terms and definitions that were used during the workshop (e.g., test case types). We mitigated this threat by providing definitions for all the terms used in the workshop. Another internal threat to validity is the likelihood that testers were influenced by the opinions of each other. However, since we construct our taxonomy based on a triangulated approach, we minimize the likelihood of this risk.

Construct Validity: This study builds on the assumption that there exists a dependency between code changes and test types. Nevertheless, there is a chance that such a dependency does not exist and that what we found was coincidental. We minimize this risk by constructing the taxonomy from the viewpoints of practitioners.

5.7 Conclusion and Future Work

The taxonomy presented in this paper aims at classifying dependencies between categories of code changes and test case types. Exploring these dependencies can potentially contribute to the improvement of ML based test case selection approaches that use code analysis and test execution results. In this paper, we have observed strong dependencies between two categories of code changes and seven test case types. This knowledge can gear the test orchestration efforts by pinpointing and executing test cases that are in relation with the relevant changes in the source code. The strongest dependencies were captured between performance, load, stress, soak, volume and the two categories of code changes: memory management and complexity. On the opposite end of the spectrum, the weakest dependencies were found between smoke, back-to-back, installability, accessibility, portability, compatibility, and backup and recovery tests, and the two categories of code changes. Those test cases can be excluded from the suite when the tested code contains memory management and complexity changes only. As a future work, we plan to continue working on refining the presented taxonomy by investigating additional dependency patterns between other test case types and categories of code changes. Another important future work is to investigate potential dependency links between test script constructs and test execution outcomes of different types. Finally, we aim at evaluating the taxonomy presented in this study by using utility demonstrations on different software projects and programming languages.

Chapter 6

Paper E

Improving Software Regression Testing Using a Machine Learning-Based Method for Test Type Selection

Al-Sabbagh, K.W., Staron, M., and Hebig, R.

In Product-Focused Software Process Improvement: 23rd International Conference, PROFES 2022, Proceedings (pp. 480-496).

Abstract

Since only a limited time is available for performing software regression testing, a subset of crucial test cases from the test suites has to be selected for execution. In this paper, we introduce a method that uses the relation between types of code changes and regression tests to select test types that require execution. We work closely with a large power supply company to develop and evaluate the method and measure the total regression testing time taken by our method and its effectiveness in selecting the most relevant test types. The results show that the method reduces the total regression time by an average of 18.33% when compared with the approach used by our industrial partner. The results also show that using a medium window size in the method configuration results in an improved recall rate from 61.11% to 83.33%, but not in considerable time reduction of testing. We conclude that our method can potentially be used to steer the testing effort at software development companies by guiding testers into which regression test types are essential for execution.

6.1 Introduction

Modern software development projects evolve rapidly as software engineers add new features, fix faults, or refactor code smells. To prevent faults from breaking existing functionality in the evolving system, software engineers frequently perform software regression testing. A safe and straightforward approach to perform regression testing is to execute a pre-defined set of test cases, usually a selection of unit, system and function tests. Such an approach is often referred to as a *retest-all strategy*. Despite benefits in set-up time, this strategy does not take into consideration changes done to the system – these are often tested during system or function test phases. The frequent execution of these retest-all test suites can also be extremely time and resource consuming. As a remedy to this, a number of Test Case Selection (TCS) methods have been developed, e.g., selecting tests based on their relevance to the modifications made in the SUT (System Under Test), in this way reducing the time and cost of testing.

A recent family of approaches for TCS employs statistical models to predict test cases based on their historical verdicts [58], [78], and [113]. These approaches are based on the assumption that a dependency between code changes and test case execution results (pass/fail) exists. For example, Knauss et al. [58] proposed an automatic recommender that analyzes the frequency in which test cases fail on a particular day given code changes made to software modules, achieving 78% reduction in the studied test suite. Similarly, one of the first implementations of ML for test case selection was presented in our previous work [78], where we introduced a TCS method that utilizes textual analysis and a conventional tree-based model to predict test case execution results.

All of these approaches create a prediction model using historical test execution results and the code changes against which these tests were exercised. However, there are several inherent challenges to the application of these approaches. One of the major challenges is the need to develop a database of source code changes over time and a database of the related test case verdicts.

In response to this challenge, we have been investigating strategies for applying test selection without the need to use historical information about test execution results. Instead of the historical verdicts, we focus on the relation between types of code changes (e.g., including new conditional statements) and type of regression tests (e.g., statement test, [55]). We have constructed a facet-based taxonomy of dependencies between code changes and test cases of specific types [140]. The knowledge presented in the dependency taxonomy is used to instrument tools for TCS by only analyzing the content of code changes, and thus determine which set of test types will be affected by the change. We address the following research question:

> *How to reduce the time of regression testing by selecting only the most relevant test types?*

We address this question by developing a machine learning-based method (and a tool) – HiTTs (*Human-in-the-loop Approach for Test Type Selection*)– that automatically identifies types of code changes and then selects the relevant types of tests. By using test types and source code changes types, we do not require historical data about the test case verdicts and therefore, HiTTs can be used already from the start of software development.

We work closely with a large power supply provider that develops software solutions to revise the taxonomy and validate the results. To evaluate our method, we used an embedded system that is owned and developed by our industrial partner. The results of this study show that our method has promising potentials in reducing the regression testing time at a high fault detection rate.

6.2 Related Work

Previous studies have been conducted to examine SW regression testing approaches, as surveyed in [141] and [68]. The majority of these approaches differ from our approach in their used artifacts (i.e., they require information about test cases) and the underlying concepts (i.e., they operate on a test level granularity). Unlike our approach, existing approaches require updating dependency graphs or coverage information to select or prioritize tests. In this section, we discuss some of these approaches and report their time usage of testing.

Greedy-based Chi et al. [142] proposed an algorithm that traces method call sequences under each test case to construct a call graph. The call graph is then used to sample the testing order based on method call sequence coverage criterion. The method was valuated and compared for effectiveness in terms of of fault detection and time usage against 22 state-of-the-art techniques. The results showed that the algorithm outperformed the other 22 techniques in terms of fault detection, but not in time usage. Specifically, the proposed technique was found to take 20.5% more than the next best technique compared for effectiveness.

Similarity-based De Oliveira Neto et al. [143] conducted a case study to investigate the efficiency of 3 similarity-based approaches for test selection, namely, the Normalised Levenshtein, Jaccard Index, and Normalised Compression Distance. The results showed that using the Normalized Levenshtein and Jaccard Index outperformed the Normalised Compression Distance in terms of their coverage rate of test requirements, dependencies, and steps. Specifically, the Normalized Levenshtein reduced the amount of executed tests by 65% and could still cover distinct combinations of test dependencies required to execute test cases. In terms of the saved time, the results showed that the Normalized Levenshtein reduced the testing time by 15.1% compared to random selection.

ML-based: Bertolino et al. [144] proposed an approach that seeks to find transitively dependent classes on changed ones in new versions of the SUT along with their associated test classes. The approach prioritizes the selected tests based on several code and test metrics which then get fed into an ML model for training. The evaluation of the approach was done by comparing 10 ML algorithms using 6 Java projects in terms of the Rank Percentile Average metric and the sum of time required for ranking the selected tests. The results showed that using the MART algorithm outperforms the others in terms of Rank Percentile Average, whereas the Coordinate ASCENT performed best in terms of ranking time.

Graph-based: Orso et al. [145] proposed a two-phase algorithm that builds a graph representation of the SUT and then identifies, based on information on changed classes, the parts of the SUT that need to be tested. As a result, tests that traverse the changed parts of the SUT are selected for execution. The authors compared the regression testing time of their approach against a retest-all baseline on 3 programs. The results showed that using their approach reduced the regression testing time between 5.9% to 89.7%, with an average of 42.8%.

6.3 Core Concepts and Background

This section presents core concepts and defines several types of code changes and tests presented in [140].

6.3.1 Core Concepts

We use the definition of a software program *P* to be a collection of lines of code $L <L_1,\dots,L_n>$. *P'* denotes a modified revision of *P*, and includes one or more combinations of added/removed/modified *L*, distant from *P*. We use the term 'revision' to refer to a modified version of *P*. A test case, denoted by *tc*, is a specification of the inputs and expected results to verify that *P'* complies without issues. The result of executing a *tc* is referred to as 'test case verdict' (passed or failed) and is denoted with *te*. A set of test cases $T = <tc_1,\ tc_2,\ \dots>$ is the test suite for testing *P'*. Regression test selection refers to the strategy of testing that given a *P'* selects a subset of *tc* that is crucial for execution.

6.3.2 The Dependency Taxonomy

The method presented in this study is based on the knowledge depicted in the dependency taxonomy that we created in collaboration with SW testers from the industry [140]. Each branch in the taxonomy refers to a single dependency between a test and a code change type, where a dependency means that a change in a code type results in a failure of tests of a specific type. In this study, we utilize and validate 8 dependency links from the original taxonomy since our industrial partner could only provide us with information about 4 test types.

Figure 6.1 illustrate the 8 dependency links between the test and code types. All dependencies in the original taxonomy were identified from two sources of information - SW testers and literature studies. The solid connectors in Figure 6.1 correspond to dependencies that were identified by testers, whereas dotted lines correspond to dependencies that were identified from the literature. Table 6.1 summarizes the definitions of the 4 test types depicted. We refer the reader to [140] for more details about each type of code and test.

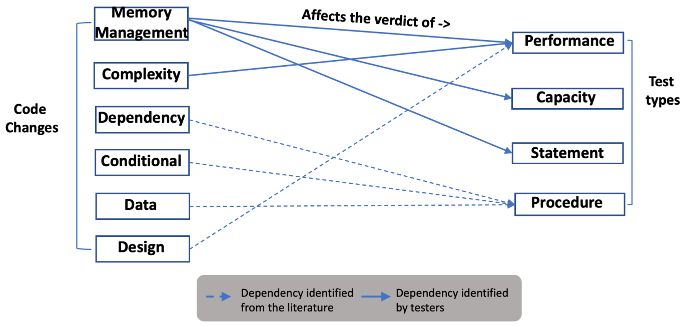


Figure 6.1: The taxonomy of dependency between code and test types.

`Memory management:` This category concerns the management of system memory during run-time. Changes in this category include introducing/fixing memory leaks, buffer overflow, dangling pointers, and resource interference with multi-threading.

`Design:` This category involves changes that include code refactoring, adding or removing methods, classes, interfaces, enumerators, or code smells.

`Complexity:` This category represents changes that add to or reduce the time complexity of the SUT. It includes changes such as adding or removing loops, conditional statements, nesting blocks or recursive functions.

`Dependency:` This category describes a change where a dependency between a module/fragment/library is added/modified. It can be importing a new library, a namespace, or a class.

`Conditional:` This category occurs when a logical operator or in a conditional statement is added/modified.

`Data:` This category involves changing 1) function parameters, 2) value assignment to variables, 3) casting, 4) array allocations, or 5) declaring variables.

Table 6.1: Definitions of test case types in the taxonomy of dependency.

Test Case Type	**Definition**
Statement	Constructed to force execution of individual statements.
Performance	Evaluate the degree to which a test item accomplishes its designated functions within a given time.
Capacity	Evaluate the level at which increasing load affects a test item's ability to sustain required performance.
Procedure	Evaluate whether procedural instructions for interacting with a test item to meet user requirements.

6.4 Research Design

In this section, we describe how our method was designed and implemented.

6.4.1 HiTTs Implementation

The basic idea behind HiTTs is to utilize the relations presented in the dependency taxonomy for automating the classification of *L* in *P'* into several code types, and as a result select regression test types that are sensitive to the changes introduced in the code. To achieve this, we use a three-phase process, which we call 1) Annotation and Training, 2) Calibration, and 3) Selection.

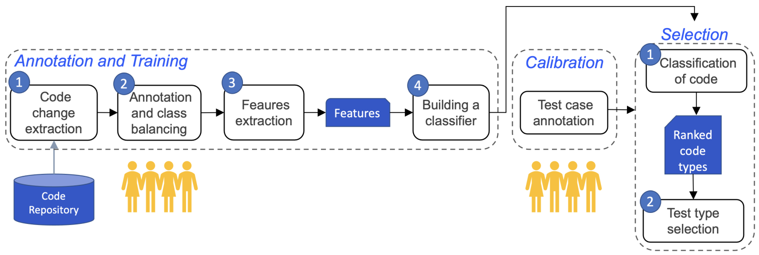


Figure 6.2: Human in the loop for test type selection.

6.4.1.1 Annotation and Training (Phase 1):

The first phase in HiTTs consists of 4 steps that concern the extraction of code changes, annotation and class balancing, features extraction, and building a classifier. A step-by-step description of each step in this phase is as follows:

1. **Code change extraction:** the method starts by extracting historical code changes between pairs of consecutive versions of *P*, *P'* from the version control system (e.g., git). Only modified and added *L* at different *P'* are retained, whereas all deleted *L* are discarded since the scope of this study is to identify regression tests that will react to new/modified code changes. The tool then parses the content of extracted *L* and filters out all *L* that belong to configuration files (e.g., .xml and .json), comments, empty and unit test *L*. Once the filtering step is complete, we save the extracted set of *L* for each *P'* in a 'csv' file.

2. **Annotation and class balancing:** each *L* in the 'csv' file is then annotated by two or more SW architects into one of the 6 categories of code change. *L* that are annotated with the same code types are retained, and the remaining ones get discarded. Once the annotation is complete, we inspect the distribution of instances under the 6 code types. If the number of instances in one code type heavily outnumbers those in the other types, we oversample instances in the minority class to balance out the data. This activity is necessary to mitigate the effect of introducing a classification bias toward one of the classes [146].

3. **Features extraction:** In this step, we transform the collected revision files into feature vectors using the bag of words (BoW) approach. We use a tool that, for each *L* in the collected revision files:

- creates a vocabulary for all L (using the BoW technique, with a cut-off parameter of how many words should be included[1])
- creates a token for words that fall outside of the frequency defined by the cut-off parameter of the bag of words
- finds a set of predefined keywords in each line,
- checks each word in the line to decide if it should be tokenized or if it is a predefined feature.

The output of this step is a large array of numbers, each representing the the token frequency of a specific feature in the bag of words space of vectors.

4. **Building a multi-class model:** the final step in this phase concerns feeding the set of extracted feature vectors into a multi-class ML model for training.

6.4.1.2 Calibration (Phase 2):

In order for HiTTs to select crucial regression test types for execution, it requires knowledge about the types of *tc* that are available in the suites. Thus, the pool of *tc* from which HiTTs can operate must include information about the type of the *tc*. This requires SW architects to calibrate the type of test in every new/existing *tc*. This can be done, for example, by creating a variable in each test class and use it to tag/calibrate the test type of the *tc*. Note that this step can be performed independently from phase 1.

6.4.1.3 Selection (Phase 3):

The final phase of HiTTs concerns the analysis of code types that are found in new code revisions and selecting regression test types that are sensitive to the changes. The phase can be described in two steps:

1. **Classification of lines of code:** The first step in this phase utilizes the trained model in phase 1 for classifying L that appear in *P'* into one of the 6 code types. As soon as the classification is complete, the method measures the count of L under each code type and generates a list of *ranked code types* from highest to lowest in terms of L count.

2. **Test type selection:** The next step in this phase is to select regression test types that are important for execution. For this, HiTTs uses a set of predefined rules that specify which test types are sensitive to what code types. These rules are derived from the taxonomy of dependency (see Section 6.3.2) and their usage is determined by a window size. The larger the window size, the more code types that HiTTs will use for

[1]BoW is essentially a sequence of words/tokens, which are descendingly ordered according to frequency. This cut-off parameters controls how many of the most frequently used words are included as features – e.g. 10 means that the 10 most frequently used words become features and the rest are ignored.

selecting test types. Specifically, HiTTs will select all test types that are in dependency with the code types that fall within the window boundary. For example, a window size of 1 would trigger HiTTs to select test types that are in dependency with the first top ranked code type only. Since the taxonomy of dependency consists of 6 code types, HiTTs can currently utilize a window size between 1 and 6.

Note that the first phase of HiTTs needs to be performed only once for training the classifier. Similarly, the second phase is performed only when a new *tc* is created or existing *tc* requires calibration.

6.4.2 Usage Scenario

In this section, we describe a usage scenario to show how test orchestrators can use HiTTs. The scenario assumes that the first and second phases of HiTTs were performed and a classifier was already built.

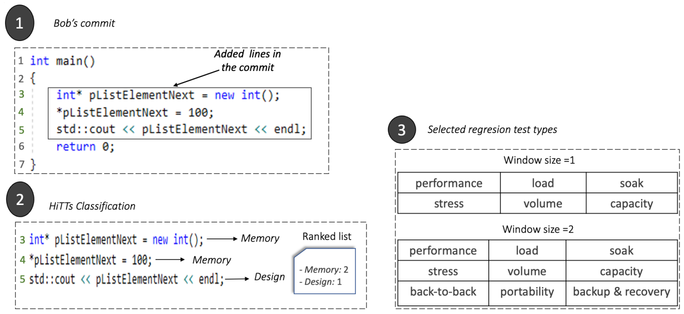
Suppose that Bob is a SW architect who is modifying a feature in the SUT. After Bob concludes his implementation, he commits his code changes to the development repository (step 1 in Figure 6.3). Lines 3, 4, and 5 in Figure 6.3 corresponds to the modified L that Bob submitted in his commit. At this point, HiTTs will analyze Bob's commit by classifying each L into one of the 6 categories of code changes (step 2 in Figure 6.3). After classifying the modified/added L, HiTTs will measure the count of classifications made with respect to each code type and accordingly generates a ranked list of code types based on their lines' count. In this example, HiTTs classified two-third of the L in Bob's commit as memory management related and one-third as design. Assuming that the test orchestrators at Bob's company set the window size of HiTTs to 1, then HiTTs will decide to select regression test types that are in dependency with the memory management code only (step 3 in Figure 6.3). As a result, performance, load, soak, stress, volume, and capacity tests will be executed to test Bob's commit. Now suppose that the test orchestrators at Bob's company decide to change the window size of HiTTs to 2. In this case, HiTTs will decide to select test types that are dependent on both memory management and design code types. Consequently, performance, load, soak, stress, volume, capacity, back-to-back, portability, and backup and recovery tests will be executed.

6.5 Evaluation of HiTTs

In this section, we present the evaluation results of our method.

6.5.1 Annotation and Training (Phase 1)

This study was performed over a period of two weeks at a large power supply provider organization that develops software solutions for managing energy



```
1 int main()
2 {
3   int* pListElementNext = new int();
4   *pListElementNext = 100;
5   std::cout << pListElementNext << endl;
6   return 0;
7 }
```

```
3 int* pListElementNext = new int();
4 *pListElementNext = 100;
5 std::cout << pListElementNext << endl;
```

Window size =1

performance	load	soak
stress	volume	capacity

Window size =2

performance	load	soak
stress	volume	capacity
back-to-back	portability	backup & recovery

Figure 6.3: An illustrative example of a usage scenario for HiTTs.

consumption in different products. The organization provided us with access to a data-set that belonged to an embedded system written in the C++ language.

6.5.1.1 Code Change Extraction (step 1)

In this study, a total of 9 code revisions were extracted from the SUT repository. We restricted the extraction of revisions to 9, since we were mainly interested in understanding the effectiveness of our method in reducing the regression testing time. The extracted revisions comprised a total of 2,103 modified and added L from which 1,321 L belonged to source code files ('.cpp' and '.h') [2].

6.5.1.2 Annotation and Class Balancing (step 2)

Five SW architects that work at the collaborating company were employed to perform the annotations. First, we organized a workshop with the SW architects, where we began by presenting definitions and code examples for each code type in the dependency taxonomy. This was necessary to ensure that all architects posses a good understanding of each type of code change in the dependency taxonomy before starting the annotation. At the end of the workshop, architects were asked to individually annotate each L in the 9 revision files and to send us the annotated L. After receiving the annotated L, we filtered out L that were not mutually annotated by the 5 architects and retained L that were annotated with the same code types. In total, we found 523 L in the annotated files to be similar in their annotation values (level of agreement = 40%). While a common rule of thumb in the literature demands a higher level of agreement between annotators, several studies have shown that comparing annotations by independent and multiple annotators can yield agreement rates as low as 22% [147] and [148].

Figure 6.4 shows the distribution of code types in the set of annotated L. The Figure shows that the majority of L belonged to the 'Design' code type (25%), whereas the minimal count of L belonged to the 'Conditional' type (4%). The 'Other' category is used by the annotators when encountering L

[2]Due to non-disclosure agreements with our industrial partner, our data-set can not be made public for replication

that does not belong to the 6 code categories. Since the distribution of code types is imbalanced, we decided to use the SMOTE module available in the Scikit-learn library [76] to balance the distribution of L in the code types. Applying SMOTE to the data-set resulted in oversampling instances in the minority code types to the same number of instances in the 'Design' code type (the majority class). As a result, we retrieved a total of 903 annotated L.

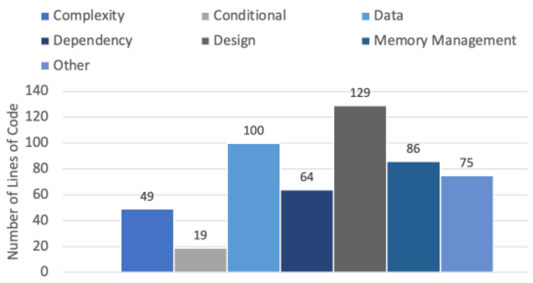


Figure 6.4: The distribution of code types in the annotated lines.

6.5.1.3 Features Extraction and Building the Classifier (step 3)

HiTTs employs a textual analysis technique that extracts features from the set of annotated code, where each feature corresponds to a code token that appears in the input file. In this study, we employ the tool proposed by Ochodek et al. [42] to perform the features extraction using the BoW model. Applying BoW on the annotated set of L resulted in a multi-dimensional array that consisted of 895 feature tokens.

HiTTs employs a multi-class classifier that classifies L into one of 6 code change types. This study employed a random forest (RF) model as the multi-classifier in HiTTs. Our choice of using RF was mainly due to the promising potentials that it showed in our recent series of publications (e.g., in [78]). In this study, the hyper-parameters of the RF model were kept in their default state as found in the scikit-learn library (version 0.20.4). The only alteration that we made was in the n_estimator (the number of trees) parameter, where we changed its value from 10 to 100. This was a design choice that we adopted based on our findings in [78], where we experimented the use of an RF model for TCS without tuning the model's parameters. Our findings showed that using untuned parameters in RF would yield better predictive performance for TCS than the other four deep-learning and tree-based models.

6.5.2 Calibration (Phase 2)

In this study, we decided to calibrate tests whose execution verdicts changed from one state to another (e.g., from 'passed' to 'failed'), at least once, during the last six months from the time of conducting this study. This was done to maximize the probability of working with tests that are sensitive to changes in the code-base. As a result, information about 868 tests were extracted from the test logs of the SUT. Architects were required to jointly agree on an ISO test type [55] that best describes each extracted test, and then use that test type for annotation. The keyword 'Other' was used by the architects to annotate tests whose specifications do not match the description of any ISO test types. Four distinct test types were used for annotating the 868 tests. The distribution of the annotated tests was as follows: procedure tests had the highest proportion with a total of 546 tests (62.9%); statement tests had the second highest proportion with 302 tests (34.7%); performance and capacity tests had the lowest proportion with one test respectively; 18 tests (2%) were annotated with the 'Other' keyword. We discarded all tests that were annotated with 'Other', as we do not know which types of code changes would trigger these tests to react.

6.5.3 Selection (Phase 3)

To evaluate the effectiveness of HiTTs, we extracted code changes committed to the SUT repository and their associated test information after the time of performing the annotation and training phase. A total of 9 code revisions and 26,576 executions of the 868 calibrated tests were extracted. Each code revision was fed into the trained RF model for classifying L into their relevant code types. Figure 6.5 shows, for each code revision, the number of classified L under each category of code type. All L that were classified as 'Other' by the model were removed from the next step of the selection phase. For the remainder of this paper, we refer to these revisions as 'evaluation revisions'.

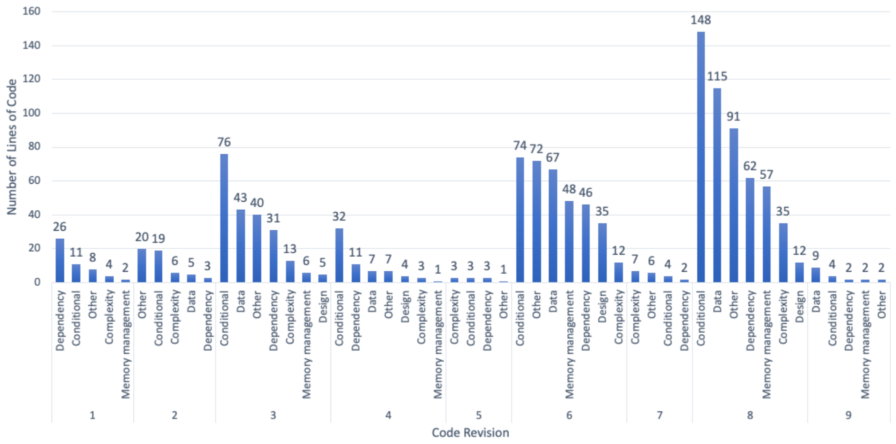


Figure 6.5: The distribution of code types in the evaluation revision.

6.5.4 Baseline Construction

To understand whether our method is effective in reducing the regression testing time, we needed to measure and compare its performance against one or more baseline measures. For this purpose, two baselines were used in this study - the actual and retest-all. The actual baseline is a measure of the total time taken to execute all *tc* that we calculated from the test logs of the build server of the SUT. The retest-all baseline is a measure of the total time taken to execute all available *tc* under the four test types in similar ratios.

6.5.4.1 Actual

Table 6.2 summarizes the information of the execution times of the four test types. The Table shows, for each revision, the number of non-commented lines of code (column 2). The 'actual baseline' column corresponds to the total *te* time taken to execute all *tc* of the four test types, as found in the test log files of the SUT. Total execution times spans from 0.91 hours to several days. Columns 4 to 7 summarize the actual *te* for capacity, procedure, statement, and performance tests respectively, whereas columns 8 to 12 show the number of *te* performed for each test type. By observing the number of *te* under each test type, we notice that not all test types are executed against the majority of the evaluation revisions. For example, capacity *tc* were only exercised against revisions 1, 2, 3, 5, 8 and 9 (as denoted with '-' in the 'nu. Capacity' column).

Table 6.2: Information about the actual test execution (in hour) for every revision.

Revision	Lines of code	Actual baseline	Capacity	Procedure	Statement	Performance	nu. Capacity	nu. Procedure	nu. Statement	nu. Performance	nu. Others
1	51	5.73	0.02	3,25	1.77	0.00	5	510	260	-	45
2	53	20.66	0.02	11.02	8.50	0.03	5	1990	1320	5	80
3	201	87.57	0.06	58.23	25.23	0.09	14	9366	1320	14	266
4	65	20.46	0.00	11.68	8.09	0.00	-	2030	2281	-	42
5	10	7.67	0.02	4.08	3.03	0.00	4	816	468	-	44
6	354	22.10	0.00	15.52	5.93	0.00	-	2303	882	-	42
7	19	0.91	0.00	0.91	0.00	0.00	-	192	-	-	-
8	520	12.08	0.03	5.36	6.00	0.00	8	1040	936	-	56
9	19	1.14	0.00	1.14	0.00	0.00	-	240	-	-	-

6.5.4.2 Retest-all

Since we do not have information about the actual *te* times of every test type across all revisions, we needed to normalize the *te* times of capacity, performance, procedure, statement in order to simulate a retest-all scenario on the 9 evaluation revisions. The normalized *te* times for capacity, performance, procedure, statement tests are presented in Table 6.2 using the following procedure. First, we calculate the average time required to execute a *tc* under each type in every evaluation revision. Second, for each evaluation revision, we subtract the number of executed tests from those executed against the revision with the highest number of *te*. Third, we multiply the number of missing *te* under each test type with the average *te* time for the same test type. Finally, we add the *te* time of the estimated *tc* to the actual *te* time that we found in the test log data. The advantage of retaining the actual *te* time of existing tests lies in minimizing the probability of using over/under-estimated *te* time. Table 6.3 summarizes the normalized *te* times for all test types across the 9 evaluation revisions. The 'Retest-all baseline' column in Table 6.3 corresponds to the total *te* time calculated by summing up the normalized values under each test type.

Table 6.3: Normalized execution times (in hour) for each test types in all revisions.

Revision	Retest-all baseline	Capacity	Procedure	Statement	Performance
1	70.63	0.06	56.51	10.09	0.09
2	67.81	0.06	55.38	8.50	0.08
3	87.57	0.06	58.23	25.23	0.09
4	67.97	0.06	55.80	8.09	0.09
5	69.12	0.06	55.51	9.72	0.09
6	71.40	0.06	58.00	9.37	0.09
7	70.44	0.06	56.09	10.35	0.09
8	68.32	0.06	55.44	9.01	0.09
9	70.38	0.06	56.03	10.35	0.09

6.5.5 Results and Analysis

The goal of the evaluation is to identify the total amount of reduced time in performing regression testing. To that end, we compare the testing time required by HiTTs with the two baseline measures. We use a window size of 1, 2, and 3 {*w1, w2, w3*} respectively for the comparison. Results of applying HiTTs with each window size are depicted in Table 6.4. The Table shows, for each revision and window size, the types of selected *tc* (column 3), the actual failing test types (column 4), the actual *te* time for all *tc* (column 5), the *te* time of a retest-all approach (column 6), the amount of reduced time relative to the retest-all (column 8) and the actual baseline (column 10) time.

The results reported in Table 6.4 suggest that using any window size in HiTTs reduces the total testing time by more than eight hours across the majority of evaluation revisions. The total reduced time, as measured by correct deselection of passing test types, reached 52.94% when compared with the actual baseline. Similarly, the percentages of improvement in time reduction relative to the retest-all baseline reached between 0.18% and 15.78%. This reduced time can potentially save architects the hurdle of doing large code rework after testing, since bugs found earlier in the development cycle are often easier to fix than bugs found after the time of adding new code. For instance, applying HiTTs with a window size of 1 on revision 8 was found to reduce the testing time by 6.03 hours compared with the actual baseline. Hence, instead of waiting for 11.4 hours to execute integration and system level tests, architects will wait for 5.37 hours to receive feedback about their code. This allows architects to spare 6.03 hours for bug fixing, feature development, or executing other types of test suites. Further, by comparing the values in the 'selected test types' and the 'failing test types' columns, we notice that the selection rate of fault-revealing tests was best when using *w2* in HiTTs.

However, what stands out in the results is that statement and capacity test types were only selected once for revision 6 when using *w3*. This can be due to missing dependency links in the taxonomy or code types. Hence, future work need to investigate additional dependencies between the capacity and statement tests, and existing code types.

To gain a better understanding of the method's effectiveness, we measured its fault detection capability in terms of recall and precision when using the three window sizes. While precision is the proportion of correctly identified test types, recall is the proportion of relevant test types that were identified as such. Having both precision and recall high ensures the detection of larger amount of test types that will reveal faults in the SUT. Further, we calculated the mean reduced time by HiTTs using the three window sizes and compared the results with the two baselines. Figure 6.6 shows a bar chart that depicts the results of the comparison. The results indicate that using *w2* or *w3* improves the rate of faults detection by 22.2% compared to when using *w1* (recall improvement from 61% to 83.33%). Conversely, the precision rate remained unchanged for *w2* (77.78%) and dropped to 69.44% for *w3*. Taken together, these results suggest that using *w3* leads to the least effective performance of HiTTs, whereas *w2* yields the highest performance. On the other hand, the mean reduced times attained when using *w1* or *w2* was found to be similar, which implies that using either of the two window sizes leads to a similar reduction rate of the testing time.

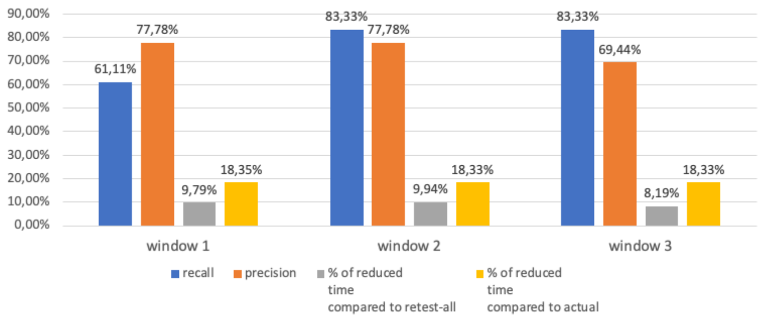


Figure 6.6: Mean performance and reduced testing time using three windows.

RQ. How to reduce the time of regression testing by selecting only the most relevant test types?
The results confirm that using our method with a window size of 2 reduces the time of regression testing by 9.94% on average compared to a retest-all approach and by 18.33% compared to the testing approach adopted by our industrial partner.

6.6 Threats to Validity

We use the framework in [46] to discuss the limitations of our paper.

Table 6.4: The evaluation results of HiTTs compared to two baselines.

Window size of 1, 2, and 3									
Revision	window	selected test types	failing test types	actual baseline	retest-all baseline	reduced time (retest-all)	% of reduced time (retest-all)	reduced time (actual)	% of reduced time (actual)
1	w1	Procedure	Procedure, Statement	5,04	66,74	0,14	0,22	0,02	0,40
	w2	Procedure	Procedure, Statement	5,04	66,74	0,14	0,22	0,02	0,40
	w3	Procedure, Performance	Procedure, Statement	5,04	66,74	0,06	0,08	0,02	0,40
2	w1	Procedure	Procedure, Statement	19,56	64,02	0,14	0,22	0,05	0,24
	w2	Procedure, Performance	Procedure, Statement	19,56	64,02	0,06	0,09	0,02	0,10
	w3	Procedure, Performance	Procedure, Statement	19,56	64,02	0,06	0,09	0,02	0,10
3	w1	Procedure	Procedure, Statement	83,61	83,61	0,15	0,18	0,15	0,18
	w2	Procedure	Procedure, Statement	83,61	83,61	0,15	0,18	0,15	0,18
	w3	Procedure	Procedure, Statement	83,61	83,61	0,15	0,18	0,15	0,18
4	w1	Procedure	Procedure	19,77	64,04	8,24	12,86	8,09	40,93
	w2	Procedure	Procedure	19,77	64,04	8,24	12,86	8,09	40,93
	w3	Procedure	Procedure	19,77	64,04	8,24	12,86	8,09	40,93
5	w1	Performance	Procedure	7,13	65,37	9,77	14,95	3,05	42,76
	w2	Performance, Procedure	Procedure	7,13	65,37	9,77	14,95	3,05	42,76
	w3	Performance, Procedure	Procedure	7,13	65,37	9,77	14,95	3,05	42,76
6	w1	Procedure	Procedure	21,45	67,51	9,51	14,09	5,93	27,65
	w2	Procedure	Procedure	21,45	67,51	10,50	15,55	5,93	27,65
	w3	Procedure, Performance, Capacity, Statement	Procedure	21,45	67,51	0,00	0,00	0,00	0,00
7	w1	Performance	Procedure	0,91	66,59	10,41	15,63	0,00	0,00
	w2	Performance. Procedure	Procedure	0,91	66,59	10,41	15,63	0,00	0,00
	w3	Performance. Procedure	Procedure	0,91	66,59	10,41	15,63	0,00	0,00
8	w1	Procedure	Procedure	11,40	64,60	9,16	14,17	6,03	52,94
	w2	Procedure	Procedure	11,40	64,60	9,16	14,17	6,03	52,94
	w3	Procedure	Procedure	11,40	64,60	9,16	14,17	6,03	52,94
9	w1	Procedure	Procedure	1,14	66,53	10,50	15,78	0,00	0,00
	w2	Procedure	Procedure	1,14	66,53	10,50	15,78	0,00	0,00
	w3	Procedure	Procedure	1,14	66,53	10,50	15,78	0,00	0,00

External validity We evaluated the effectiveness of HiTTs on 9 revisions that belong to a single industrial system. Thus, we cannot claim that our findings generalize well to other types of systems. However, we increase the likelihood of drawing a representative sample by using all revisions that were committed to the development repository after the time of building HiTTs. Further, we trained the classifier in HiTTs on a small sample of data, which could have resulted in a lower classification performance than what we could achieve with a larger sample. However, our evaluation shows that the performance of HiTTs is high.

Construct validity The dependency links used for defining the static rules of procedure tests were drawn from the literature, and thus not validated. However, our evaluation results showed that HiTTs was effective in selecting this type of tests across the 9 evaluation revisions.

Internal validity An internal threat is the presence of undetected defects in the tools that we used for features extraction, code change extraction, and baseline measurements. To increase our confidence in the tools' implementation, we tested our code on smaller examples. The results might differ if we employ other types of models. However, in this study we were only interested in understanding the effectiveness of HiTTs in reducing the regression testing time.

Conclusion validity There is a probability that some tests failed due to non-deterministic executions (i.e., flaky tests) or environmental factors (e.g., a hardware element goes offline). As a result, the test execution times that we used for calculating the baselines may belong to tests that failed due to factors unrelated to code changes, and thus lead us to wrong conclusions. To minimize this threat, we collected data of several thousand test executions and minimized the probability of selecting tests that have non-deterministic behaviors.

6.7 Conclusion and Future Work

In this paper, we introduced HiTTs - a machine learning based method that selects regression test types based on their relation with code types that appear in new revisions without the need of history test information. The presented method was evaluated on an industrial data-set for effectiveness in reducing the regression testing time and faults detection. The results of the study are encouraging: 1) for the subject considered, our method showed considerable time reduction in regression testing - up to 52.94%, 2) increasing the window size of HiTTs to a medium level improves the effectiveness rate of faults detection and still reduces the total time of regression testing.

The results of our study suggest several avenues for future work. First, working on refining and extending the taxonomy to capture more dependencies between the statement and capacity test types and existing code types is needed to improve the effectiveness of the method in TCS. Second, we plan to extend HiTTs by adding an ensemble of classifiers to predict the verdict of tests that belong to each selected test type. This allows HiTTs to operate on a finer-level of granularity (i.e., test case level). Finally, future work needs to compare the effectiveness of HiTTs with state-of-the-art approaches for TCS.

Chapter 7

Paper F

Predicting Build Outcomes in Continuous Integration Using Textual Analysis of Source Code Commits

Al-Sabbagh, K.W., Staron, M., and Hebig, R.

In Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering (pp. 42-51).

Abstract

Machine learning has been increasingly used to solve various software engineering tasks. One example of its usage is to predict the outcome of builds in continuous integration, where a classifier is built to predict whether new code commits will successfully compile. The aim of this study is to investigate the effectiveness of fifteen software metrics in building a classifier for build outcome prediction. Particularly, we implemented an experiment wherein we compared the effectiveness of a line-level metric and fourteen other traditional software metrics on 49,040 build records that belong to 117 Java projects. We achieved an average precision of 91% and recall of 80% when using the line-level metric for training, compared to 90% precision and 76% recall for the next best traditional software metric. In contrast, using file-level metrics was found to yield a higher predictive quality (average MCC for the best software metric= 68%) than the line-level metric (average MCC= 16%) for the failed builds. We conclude that file-level metrics are better predictors of build outcomes for the failed builds, whereas the line-level metric is a slightly better predictor of passed builds.

7.1 Introduction

Continuous integration (CI) is a modern software engineering practice in which developers integrate their code into a shared repository to enable swift detection of quality issues and bugs before releasing new features to end users [149].

A CI system typically attempts to launch a build job multiple times a day, either for each new commit submitted to the version control system or at set time intervals during the day [150]. The goal of these jobs is to notify software engineers about faults in the source code as quickly as possible in order to quickly fix them. A typical build server runs tools such as compilers and static analyzers to detect styling and quality related problems in the code that get reported to developers. Failures produced by any of these tools result in a build failure.

The completion of build jobs in a fast manner directly affects the productivity of programmers [151], as they might get distracted by other tasks while waiting for the build job to finish. As a consequence, the number of code changes committed by developers during a day will be reduced. For this reason, keeping a high pace of the build job, and understanding the root cause of build failures is key to improve the development productivity. In fact, a previous analysis on the TravisTorrent database image, created on February 8, 2017, showed that the median time to build Java projects took over 900 seconds (15 minutes) [152]. This means that developers will incur, on average, a time latency of 15 minutes before receiving feedback about their committed code from the CI environment. Therefore, reducing the time-feedback to developers is necessary to allow them to immediately start working on new development tasks with confidence that their previously committed code will pass the build phase.

To address the problem of time latency and reduced development productivity in CI, several researchers utilized machine learning (ML) models to predict the outcome of build jobs using a diversity of product and process software metrics, such as code churn size, number of commits, team size etc as features for characterizing build outcome (failed/passed). For example, Hassan and Zhang [3] mined a diversity of product and process metrics in historical projects, such as the number of modified subsystems and certification results of previous build for constructing an ML model for build prediction. Their results indicate that training a decision tree classifier on such information can yield to a correct prediction for 95% of passing builds and 69% of failing builds. Xia and Li [4] evaluated the use of nine classifiers on 20 software metrics for 126 open source projects. Their results show that using the examined metrics result in an F1-score higher than 0.7 for 21 of build outcomes. Thus, product and process metrics have shown promising results when it comes to prediction of build outcomes. In this study, we refer to these metrics by using the term traditional software metrics (TSM).

Despite these promising results of TSM-based approaches, they can only provide indications about which parts of the system, e.g. what file, causes the build to fail. However, they cannot locate the source of the failure, e.g. by indicate which lines of code might potentially cause build failures. This research aims at filling this gap. Using a textual analysis (TA) approach, we measure the frequency count of token appearances in the source code (e.g., if and while) on a line of code level. We use the term token frequency (TF) metric to refer to the measurements produced by the TA approach. However, in the context of build outcome prediction, it is unclear whether a prediction of build failure made using a line-level metric, such as TF, can be as precise and good as a prediction made using TSM, which can use per file information.

Therefore, in this study we set off to examine the effectiveness of the TF metric by empirically comparing its effectiveness against a set of TSM in build outcome prediction for 117 Java open source projects. We record the precision, recall, F1-score, and Mathews Correlation Coefficient (MCC) measures attained after training a classifier on each metric respectively. More concretely, we design and implement a controlled experiment wherein fourteen different TSM metrics extracted from the TravisTorrent [49] data-set, created on February 8, 2017, and the token frequency metric are examined for effectiveness across 49,040 builds. In our study, we address the following research question:

> *How effective is the token frequency metric in comparison with traditional software metrics for predicting build outcomes in CI?*

The specific contributions of this paper are as follows:

- we empirically investigate the effectiveness of a line-level metric in learning build outcomes in CI, and compare its effectiveness with a diversity of traditional software metrics using 117 Java based open source projects.
- we found that using TF for training a classifier slightly outperforms the effectiveness of file-level TSMs in predicting passing builds, with a small effect size difference.
- we found that using file-level metrics is more effective than TF in predicting build failures.
- we complement the TravisTorrent data-set from 2017 with a new data-set that contains TSM and TF metrics for historical code changes made in 117 Java based projects [1].

The remainder of this paper is organized as follows. Section 2 provides an overview of related work that propose approaches for CI build prediction. In Section 3 we present the experimental design and operations carried out in this study. Section 4 presents the results of our study. Section 5 discusses the threats to validity. Finally, Section 6 concludes the paper and outlines future work.

7.2 Related Work

In this section, we present related work on build outcome prediction and reasons of build failures in CI.

[1] https://doi.org/10.5281/zenodo.6784987

7.2.1 Software Metrics for Build Prediction

Several studies have proposed approaches for modelling the relationship between build statuses (passed/failed) and software metrics [153], [152], and [4]. Ni and Li [153] adopted cascaded classifiers to predict build outcomes using 18 software metrics to characterize historical build jobs for 532 Java and Ruby projects. The results showed that using 'Historical Statistic' metrics are the most useful features in predicting the build outcome with an accuracy rate of 75.3%. Hassan and Wang [152] employed a random forest classifier for predicting build outcomes using features derived from error logs in historical build records. The results of their work showed an average F-score rate of 87% in the prediction of build outcomes. Another example is the work conducted by Xia and Li [4], where the authors evaluated the performance of nine different classifiers using traditional software metrics in build predictions. Their results showed that a Decision tree, gradient boosting and random forest classifiers outperform the other algorithms in F1-score, achieving a 17% more F-score on average. With these classifiers, build outcomes for a quarter of the analyzed projects can be predicted with F1-score over 60%. In another empirical study conducted by Xia et al. [154], the authors evaluated the performance of six classifiers for build outcome prediction. The results of their study revealed that a Decision Tree classifier performs the best in comparison with the other five classifiers with a score of 17% for F1-score on average.

Despite the promising results that the majority of these studies achieved, non of them has investigated the effectiveness of metrics that operate on a line of code level for build prediction. In this paper, we characterize historical changes of source code on a line-level of abstraction and analyze its effectiveness in predicting build failures.

7.2.2 Reasons of Build Failure in CI

Over the recent years, studies on identifying factors that result in build failures are increasingly gaining more attention by researchers [155], [150], [156]. Rausch et al. [155] investigated factors that result in build failures. The findings drawn from the analyses of historical build logs suggest that failing integration tests, code quality, and compilation errors are the most common factors that lead to build failures. Luo et al. [156] used the TravisTorrent data-set to investigate factors that cause build failures. They found that in our study, the number of commits in a build (git_num_all_built_commits) is the most important factor that has a significant impact on the build result. Beller et al. [150] conducted an in-depth quantifiable study using TravisTorrent data-set to investigate the effect of testing on build failures. The results of their work concluded that testing is the most important factor that leads to build failure. Moreover, the programming language has a strong impact on the number of executed tests, the time they take to execute, and their proneness to fail. In this paper, we expand on these empirical studies by examining the effectiveness of a new metric (token frequency) which can potentially identify the root cause of build failures in the source code.

7.3 Experiment Design and Operations

This section describes the experiment design, the data-sets, and the operations carried out to implement this experiment.

7.3.1 Data Collection and Preprocessing

TravisTorrent is a synthesized open-source data-set that consists of 2,640,825 build job records belonging to 1,300 projects (402 Java projects and 898 Ruby projects) [49]. Every build job record in the data-set synthesizes information from three data sources: The project's git repository, data extracted from GitHub, and data from Travis's API. In total, the data-set provides 55 software metric values for each historical build.

7.3.1.1 Traditional Software Metrics

In this study, we utilized the BigQuery interface for the TravisTorrent data-set, created on February 8, 2017, to mine historical build records for fourteen traditional software metrics. Table 7.3 provides a brief summary of each metric. We chose to only examine the effectiveness of these metrics as they characterize changes made to the source code (product specific) and the process, whereas the remaining metrics in TravisTorrent characterize test related aspects (e.g., tests added and tests deleted). Further all the selected traditional software metrics were previously examined in different build prediction studies such as [153] and [152].

Since the goal of this study is to evaluate the effectiveness of different software metrics in learning build outcome (pass/fail), we restrict the sample of collected historical build records and projects to fulfill the following two criteria. First, we filtered out all records whose build status (tr_build) values resolved to errored or canceled, and only kept track of those that resolved to passed/failed. Second, we only queried projects that were written in the Java programming language and included at least one failing/passing build job record. The outcome was a data-set that comprised of 117 Java projects with a total of 49,040 build records. Information about the distribution of the collected build status records and project names are summarized in columns 'Failing Builds' and 'Passing Builds' in Table 7.1.

7.3.1.2 The Token Frequency Metric

To instrument the measurement of the token frequency metric, we implemented a TA based tool that follows the procedure introduced in [98]. The procedure enacts three sequential steps that can be summarized as follows:

Step 1 (extraction of code changes): This step involves extracting code changes committed to the development repository of each analyzed project. For each project, we extract modified/added lines of code between pairs of

Table 7.1: Distribution of Build Outcomes and Lines of Code Changes in the Analyzed Projects

Id	Project	Builds	Failing builds	Passing builds	lines extracted	Failing lines	passing lines	Id	Project	Builds	Failing builds	Passing builds	Lines extracted	Failing lines	Passing lines
1	OpenRefine	192	14	178	3684	332	3352	61	picard	284	11	273	10306	378	9928
2	psi-probe	200	4	196	50471	100	50371	62	hivemall	173	17	156	20219	159	20060
3	error-prone	152	3	149	91072	26	91046	63	seyren	281	14	267	7141	136	7005
4	u2020	245	8	237	4404	135	4269	64	lenskit	274	22	252	54601	808	53793
5	metrics	279	23	256	8504	1667	6837	65	springside4	226	57	169	19121	6279	12842
6	rewrite	184	72	112	13399	3693	9706	66	onebusaway-android	187	9	178	19743	63	19680
7	checkstyle	1368	30	1338	93435	357	93078	67	rxjava-jdbc	192	2	190	7847	17	7830
8	ProjectRed	268	66	202	1075	385	690	68	core	516	6	510	75569	375	75194
9	brightspot-cms	548	62	486	8737	2753	5984	69	selendroid	445	47	398	58650	17944	40706
10	assertj-android	118	36	82	14267	3349	10918	70	nutz	924	367	557	57793	20243	37550
11	LittleProxy	287	42	245	7806	916	6890	71	jphp	300	34	266	142552	18063	124489
12	blueprints	432	127	305	41217	15106	26111	72	owner	387	7	380	20248	240	20008
13	cassandra-reaper	262	20	242	7688	1004	6684	73	twilio-java	221	8	213	28005	5547	22458
14	restlet-framework -java	436	277	159	109038	48701	60337	74	restlet-framework -java	436	277	159	109038	48701	60337
15	nodeclipse-1	238	13	225	21274	24	21250	75	azkaban	176	8	168	56976	6026	50950
16	rultor	1156	275	881	33023	10185	22838	76	nodeclipse-1	238	13	225	21274	24	21250
17	jmonkeyengine	714	9	705	69466	782	68684	77	idea-gitignore	187	45	142	26477	4148	22329
18	pdfsam	336	91	245	108882	20235	88647	78	keywhiz	240	2	238	12128	5	12123
19	robospice	74	29	45	13307	7575	5732	79	jsprit	210	4	206	20950	118	20832
20	pushy	333	21	312	4281	9	4272	80	stubby4j	571	144	427	36510	12346	24164
21	parceler	227	4	223	15645	232	15413	81	qulice	413	33	380	10885	243	10642
22	dynjs	320	20	300	32833	899	31934	82	jinjava	227	3	224	11092	8	11084
23	mybatis-3	471	15	456	94630	572	94058	83	auto	251	34	217	11912	126	11786
24	HikariCP	326	17	309	29153	3490	25663	84	xtreemfs	272	41	231	50048	2097	47951
25	thredds	333	100	233	24625	6308	18317	85	jmxtrans	400	22	378	7752	173	7579
26	maven-git-commit -id-plugin	201	31	170	14030	1499	12531	86	less4j	647	71	576	72745	7426	65319
27	dagger	302	24	278	3205	105	3100	87	cas-addons	229	7	222	7022	63	6959
28	jade4j	207	11	196	15059	265	14794	88	goclipse	228	20	208	66453	462	65991
29	jsonld-java	196	13	183	32630	76	32554	89	ccw	331	142	189	13859	3678	10181
30	webcam-capture	342	22	320	32386	227	32159	90	unirest-java	301	17	284	3558	225	3333
31	jInstagram	219	7	212	18103	242	17861	91	waffle	203	23	180	19207	160	19047
32	spring-cloud-config	251	22	229	22734	1206	21528	92	MozStumbler	517	12	505	7707	15	7692
33	gpslogger	265	36	229	14238	918	13320	93	HearthSim	234	11	223	59681	347	59334
34	jcabi-http	221	34	187	4329	596	3733	94	rexster	324	23	301	34909	464	34445
35	p6spy	333	100	233	13584	5920	7664	95	retrofit	747	5	742	17656	335	17321
36	htm.java	442	4	438	49931	413	49518	96	DSpace	1242	43	1199	77447	2099	75348
37	go-lang-idea-plugin	780	81	699	31476	1212	30264	97	structr	740	252	488	105605	58803	46802
38	Singularity	152	36	116	7302	1861	5441	98	airlift	253	123	130	21916	13609	8307
39	android	671	46	625	13857	3885	9972	99	traccar	1324	24	1300	67666	138	67528
40	jcabi-github	502	146	356	17052	5765	11287	100	querydsl	1153	194	959	33926	809	33117
41	sms-backup-plus	248	20	228	14921	195	14726	101	yobi	24	2	22	90790	3531	87259
42	truth	96	18	78	17907	1260	16647	102	openwayback	229	29	200	10976	30	10946
43	joda-time	186	5	181	18153	34	18119	103	cloudify	4137	717	3420	287810	50339	237471
44	logback	183	49	134	66311	1775	64536	104	play-authenticate	178	27	151	4823	295	4528
45	mockito	320	56	264	66687	2774	63913	105	RoaringBitmap	247	21	226	38994	685	38309
46	Hystrix	508	202	306	38633	16536	22097	106	jPOS	285	10	275	36033	148	35885
47	blueflood	744	80	664	39209	4160	35049	107	javaslang	722	8	714	384967	3997	380970
48	java-design-patterns	630	5	625	69967	51	69916	108	frontend-maven-plugin	273	27	246	2598	106	2492
49	DDT	183	62	121	55702	10375	45327	109	jodd	439	23	416	141987	1363	140624
50	dropwizard	1048	64	984	48830	1070	47760	110	quickml	222	43	179	13670	421	13249
51	nokogiri	439	117	322	23572	9520	14052	111	okhttp	1341	335	1006	64755	15756	48999
52	android-maven-plugin	224	141	83	74259	4096	70163	112	bnd	459	24	435	31434	3355	28079
53	jcabi-aspects	304	34	270	5354	858	4496	113	AcDisplay	371	187	184	31453	17569	13884
54	intellij-elixir	107	2	105	237184	952	236232	114	jedis	427	61	366	36361	878	35483
55	jsonschema2pojo	294	1	293	12985	3	12982	115	Hydra	210	35	175	6662	525	6137
56	lorsource	1470	58	1412	31970	656	31314	116	storio	192	11	181	13058	747	12311
57	analytics-android	206	17	189	6896	490	6406	117	Jest	370	71	299	22414	2460	19954
58	storm	65	36	29	28317	16622	11695								
59	basex	322	40	282	56481	1947	54534								
60	spark	249	13	236	5507	83	5424								

Table 7.2: Output From the Feature Vectors Using Bag of Words

Filename	Path	`if`	`int`	`a`	`Aa`	Content
firstFile.c	c:/folder	1	0	3	2	`if(condition==true) printf("Hello");`
firstFile.c	c:/folder	0	0	2	0	`printf("\n");`
secondFile.c	c:/folder	0	1	1	0	`int i = 10;`

consecutive builds. All extracted lines between each pair are then labeled with the execution outcome (passed/failed) of the newer build. The build execution outcomes are provided in TravisTorrent under the field 'build_status'. Thereafter, we save the extracted lines of code for every project in a 'csv' file for every analyzed project. A total of 117 csv files (one file for each project) were collected and stored locally before being processed in step 2 of the TA procedure[2].

Step 2 (features extraction): The second step utilizes a textual analysis tool [42] to convert the corpus of extracted code changes in step 1 into feature vectors. For each line of code in the collected corpus, the tool:

- creates a vocabulary for all lines of code (using the BoW technique, with a cut-off parameter of how many words should be included[3])
- creates a token for words that fall outside of the frequency defined by the cut-off parameter of the bag of words
- finds a set of predefined keywords in each line,
- checks each word in the line to decide if it should be tokenized or if it is a predefined feature.

The output of this step is a large array of numbers, each representing the the token frequency of a specific feature in the bag of words space of vectors. Table 7.2 illustrates an exemplary output of the bag of words vectors for a simple code fragment written in the C language. In this study, we chose to use a bi-gram model for representing the feature vectors, as it was previously shown to produce good learning performance in a similar context (e.g., [98]). Notice how the feature values in Table 7.2 correspond to the frequency counts of each token that appears in every line of code in the code example.

Step 3 (training a classifier): Finally, the extracted set of feature vectors from step 2 are fed into an ML model for learning how to classify new lines of code as triggering to CI builds pass/failure. The result of applying the TA technique on the collected projects resulted in extracting historical code

[2]https://anonymous.4open.science/r/CIbuilds_TSM_TF-CE19/

[3]BoW is essentially a sequence of tokens, which are descendingly ordered according to frequency. This cut-off parameters controls how many of the most frequently used words are included as features – e.g. 10 means that the 10 most frequently used words become features and the rest are ignored.

Table 7.3: Descriptions of The Examined Software Metrics

Id	Metric	Description
1	gh_num_commits_in_push	Number of commits in the push that started the build
2	git_prev_commit_resolution_status	String, "merge found" if this build is a merge otherwise "build found"
3	gh_team_size	Size of the team contributing to this project within 3 months of last commit
4	git_num_all_built_commits	Number of all commits in this build
5	gh_num_commit_comments	The number of comments on git commits on GitHub
6	git_diff_src_churn	How much (lines) production code changed by the new commits in this build
7	gh_diff_files_added	Number of files added by the new commits in this build
8	gh_diff_files_deleted	Number of files deleted by the new commits in this build
9	gh_diff_files_modified	Number of files modified by the new commits in this build
10	gh_diff_src_files	Number of production files in the new commits in this build
11	gh_diff_doc_files	Number of documentation files in the new commits in this build
12	gh_diff_other_files	Number of remaining files which are neither production code nor documentation in the new commits in this build
13	gh_num_commits_on_files_touched	Number of unique commits on the files included in this build within 3 months of last commit
14	gh_sloc	Number of executable production source lines of code, in the entire repository
15	token frequency	The frequency count of code tokens in the analyzed source code.

changes made to every collected project, as summarized in Table 7.1 under the 'Lines' column. The distribution of classes assigned to the extracted lines is specified under the columns 'Failing lines' and 'Passing lines' in Table 7.1.

7.3.2 Independent Variables

In this study, software metric is the only independent variable (treatment) examined for effectiveness on the performance of a build prediction model. A total of 15 variations (software metrics) to the independent variable were examined, as summarized in Table 7.3. Detailed description about metrics 1 to 14 can be found in the TravisTorrent database [49], whereas metric fifteen (token frequency) is a measure of the frequency count of code tokens in the analyzed programs using textual analysis.

7.3.3 Evaluation Metrics

We chose four state-of-the-art metrics to evaluate the performance of a classifier for build outcome prediction that we train on the TSM and TF metrics respectively. The four metrics are precision, recall, F1-score, and Matthews Correlation Coefficient.

While precision is the proportion of correctly identified passing builds, recall is the proportion of relevant builds that were identified as such. Having both precision and recall high ensures the detection of larger amount of passing builds and the reduction of false alarms about failing builds.

The F1-score indicates whether the predictive model is performing well in identifying builds that are actually passing (high precision) and generating little false alarms about failing builds (high recall). One drawback in using the F1-score metric is the fact that it only accounts for three elements in the confusion matrix (true positives, false positives, and false negatives), which might lead to misleading conclusions if the distribution of the binary classes in the training data is imbalanced [157].

To mitigate these drawbacks that suffice in F1-score, we decided to measure the model's MCC, which takes into account the four elements in the confusion matrix [157]. In the context of build outcome prediction, a high MCC indicates that the predictions obtained by the model are good in both classes (passing and failing builds), as MCC takes the four elements of the confusion matrix into account. Thus, MCC considers what share of the elements (builds or lines) in the failing class are correctly identified as failing. If the share is low then MCC is worse than if the share is high.

7.3.4 Experimental Hypotheses

We hypothesize that using token frequency features for constructing a predictive model is more effective in learning build prediction than traditional software metrics. The hypotheses are based on the assumption that build failures are triggered when faults in the code base are introduced. Accordingly, four hypotheses are defined and tested for statistical significance. The hypotheses are formally defined as follows:

- H_{0p}: *The mean precision is the same for a model trained on token frequency and each traditional software metrics.*

$$\mu_{1p} = \mu_{2p} \tag{7.1}$$

- H_{0r}: *The mean recall is the same for a model trained on token frequency and each traditional software metrics.*

$$\mu_{1r} = \mu_{2r} \tag{7.2}$$

- H_{0f}: *The mean F1-score is the same for a model trained on token frequency and each traditional software metrics.*

$$\mu_{1f} = \mu_{2f} \tag{7.3}$$

- H_{0mcc}: *The mean MCC is the same for a model trained on token frequency and each traditional software metrics.*

$$\mu_{1mcc} = \mu_{2mcc} \tag{7.4}$$

7.3.5 Data Analysis Methods

To decide whether to run a parametric or non-parametric statistical test for analysis, we begin the analysis by running a normality test on the distribution of the four evaluation metrics under the 15 treatment levels. We chose to use the Shapiro Wilk test to evaluate the normality of the distributions. Based on the normality test results, we decided to run the Kruskal-Wallis (a non-parametric test) for comparing the precision, recall, F1-score, and MCC values between the different treatment levels.

While the Kruskal-Wallis statistical test is used to determine statistical significance between the treatment levels and the evaluation metrics (i.e., if the treatment has an effect on precision, recall, F1-score, and MCC), they do not quantify the amount of difference between the groups. Hence, we decided to complement the analysis by calculating the effect size between the precision, recall, F1, and MCC scores attained when using the `TF` and the next best traditional software metric. For this purpose, we used the 'effsize' library available in R-studio (release 2022.02.3). We used the Cliff's Delta analysis method (a non-parametric statistical test) to measure the effect size. An effect size of +1.0 or -1.0 indicates that there is no overlap between the distribution of precision, recall, F1-score, and MCC. An effect size of 0.0 indicates that the distribution between each pair of evaluation metrics overlaps completely.

7.3.6 Prediction Model

In this study, we chose to employ a random forest (RF) model for learning build outcome prediction. This was mainly because RF 1) has a white-box nature that can be utilized to extract the set of features that influences the prediction, and 2) tends to perform well with discrete and high-dimensional input data [158]. The hyper-parameters of the model were kept in their default state as found in the scikit-learn library (version 0.20.4). The only alteration that we made was in the n_estimator (the number of trees) parameter, where we changed its value from 10 to 100. This was a design choice that we made based on the findings reported in a previous study [78] in which the authors experimented the use of an RF for predicting test case execution outcomes. The findings showed that using an RF model with the same default parameters would outperform four other deep learning and tree models in test case outcome prediction.

7.3.7 Experimental Subjects and Class Balancing

We began the experiment by applying 10-fold stratified cross-validation on the build records to create the experimental subjects. Each generated subject (fold) was used for validating the RF classifier, which we trained on the remaining nine folds for each TSM metric. Similarly, 10-fold stratified cross-validation was applied on the set of code changes that we extracted from each project.

One aspect that is known to affect the performance of predictive models is related to class imbalance, where the number of training instances in the data for one class outnumbers instances that belong to the other class [146]. The effect of training a model on imbalanced data-set lies in creating a model that is biased towards one of the classes. In order to control the effect of this aspect, we achieved a balanced distribution of build records and lines of code in the minority class of of each training fold in every analyzed project. To that end, we used the 'resample' module provided in the Scikit-learn library [76] whenever more than 50% of build records or lines of code at each project belonged to either one of the binary classes. Note that the resampling procedure was applied to the training data-sets only, as we wanted to evaluate the model's generalizability on real-world scenarios where data-sets come unbalanced.

7.4 Results

This section describes the results of the statistical tests conducted to evaluate the four hypotheses and answer the research question.

7.4.1 Evaluation of Metrics effectiveness

To evaluate the effectiveness of the TSM and TF metrics, we begin by calculating the descriptive statistics of the precision, recall, F1-score, and MCC for the RF model that we trained on each fold in every analyzed project. Table 7.4 presents descriptive statistics describing the mean and standard deviation (SD) of precision, recall, F1-score, and MCC for the total number of folds (N) in the entire set of analyzed projects. The descriptive statistics reveal that learning build prediction is most effective when using the token frequency and the `gh_num_commits_on_files _touched` metrics [4]. While the token frequency metric slightly outperformed the `gh_num_ commits_on_ files_touched` metric with respect to precision, recall, and F1-score, the `gh_num_commits_on_files_touched` metric surpassed the latter with respect to MCC. A big difference between F1 and MCC can happen if the classes in the data-set are not balanced. While we balanced the training data-set, we did not for the test data-set as explained above. Hence, the number of failing lines that are falsely predicted as passing by `TF` is fairly small compared to the lines correctly predicted to pass and incorrectly predicted to fail, a line

[4]We are aware of the wide spread in the distribution of precision, recall, F1-score, and MCC (high SD) values. Therefore, we used non-parametric statistical tests for comparing the distribution of values.

Table 7.4: Descriptive Statistics for the Precision, Recall, F1-score, and MCC

		Precision		Recall		F1-score		MCC	
Metric	**N**	**Mean**	**SD**	**Mean**	**SD**	**Mean**	**SD**	**Mean**	**SD**
gh_num_commit_comments	1170	0.38	0.35	0.48	0.49	0.348	0.353	0.03	0.08
gh_num_commits_in_push	1170	0.66	0.25	0.52	0.33	0.516	0.257	0.24	0.25
gh_num_commits_on_files_touched	1170	0.90	0.15	0.76	0.17	0.816	0.157	0.68	0.27
gh_prev_commit_resolution_status	1170	0.50	0.30	0.48	0.39	0.427	0.288	0.08	0.18
gh_sloc	1170	0.84	0.27	0.60	0.36	0.653	0.325	0.55	0.41
gh_team_size	1170	0.48	0.35	0.52	0.43	0.470	0.357	0.15	0.40
git_diff_doc_files	1170	0.36	0.33	0.47	0.48	0.354	0.350	0.03	0.10
git_diff_files_added	1170	0.62	0.27	0.58	0.41	0.495	0.287	0.19	0.20
git_diff_files_deleted	1170	0.52	0.33	0.55	0.47	0.422	0.334	0.10	0.16
git_diff_files_modified	1170	0.68	0.21	0.60	0.27	0.601	0.209	0.31	0.26
git_diff_other_files	1170	0.63	0.22	0.60	0.33	0.558	0.240	0.23	0.23
git_diff_src_churn	1170	0.82	0.18	0.67	0.22	0.715	0.176	0.52	0.26
git_diff_src_files	1170	0.68	0.20	0.61	0.26	0.614	0.198	0.31	0.25
git_num_all_built_commits	1170	0.57	0.30	0.54	0.44	0.440	0.311	0.13	0.18
token frequency	1170	0.91	0.13	0.80	0.15	0.846	0.133	0.16	0.21

that is failing is not unlikely to be predicted as passing using TF. Figure 7.1 is a bar plot that visualizes the mean scores of the four evaluation metrics for each software metric across the 117 projects. The x-axis represents the metric names, and the y-axis corresponds to the evaluation metrics' values. From the dotted frame in Figure 6.2, it can be seen that by far the greatest mean precision, recall, and F1-score were achieved when using the gh_num_commits_on_ files_touched and the token frequency metrics.

To gain a better understanding of the effectiveness of each metric, we plotted the distribution of precision, recall, F1-score and MCC values for every project. Figures 7.2(a), 7.2(b), 7.2(c) and 7.2(d) are boxplot charts that visualize the distributions. By inspecting the distributions, we observe the following:

- there exists a large disparity in the distribution of the four evaluation metrics with respect to the majority of the examined software metrics.
- the TF metric yields better MCC scores than all the other metrics in several other projects.
- the lowest attained precision, recall, and F1-score values when using TF is higher than the lowest value attained when using the other TSM metrics.

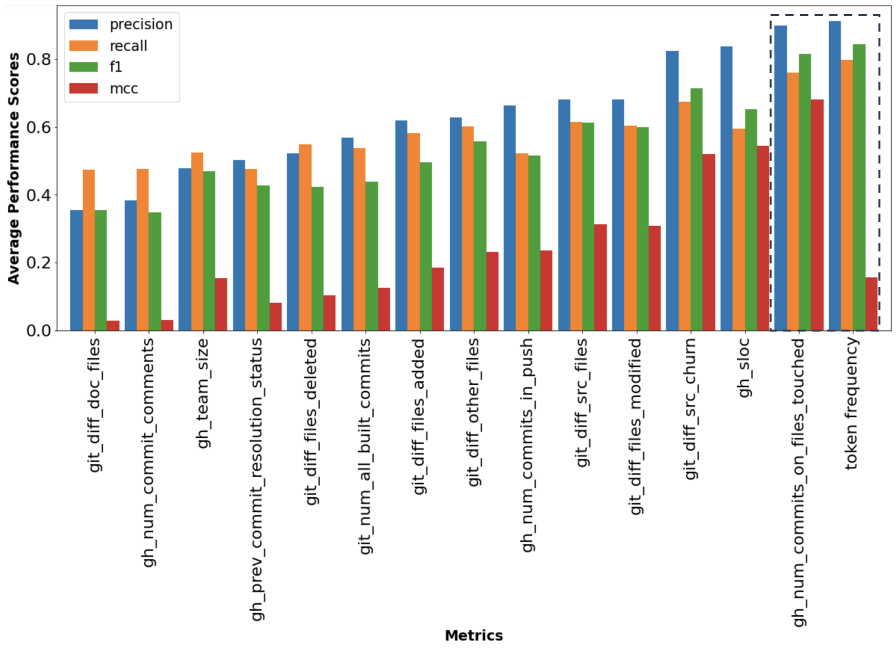


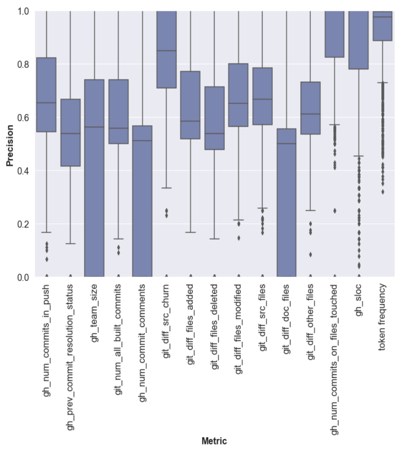
Figure 7.1: Mean Performance Scores of Each Metric.

7.4.2 Hypotheses Testing

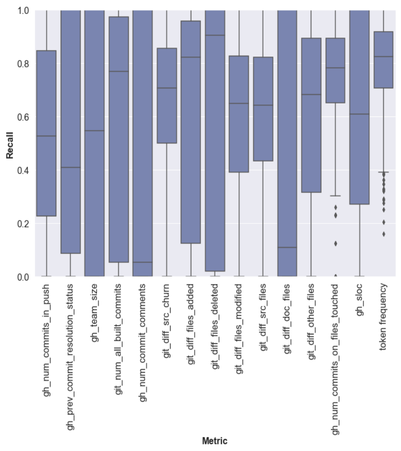
7.4.2.1 Significance Testing

We begin the hypotheses testing by conducting a normality test for the distribution of the four evaluation metrics. The statistical test results of normality for the four evaluation metrics when learning a classifier from the 15 software metrics show that the assumption of normality can be rejected for all the four evaluation metrics (p <0.05). Therefore, we decided to use a non-parametric statistical test for testing the hypotheses. To answer the research question of *How effective is the token frequency metric in comparison with traditional software metrics for predicting build outcomes in CI?*, we started the analysis by running the Kruskal-Wallis test for comparing the distribution of the evaluation metrics attained when using the TSM and TF metrics. The Kruskal-Wallis test results show that there is a statistically significant difference between the precision, recall, F1-score, and MCC variables with respect to the 15 software metrics (p-value <0.05). Table 7.5 summarizes the Kruskal-Wallis test results for each evaluation metric respectively. Since the Kruskal-wallis test is an omnibus statistical test i.e., it cannot tell which variable is statistically significantly different, we decided to run a Dunn's post hoc statistical test. To control possible family-wise error rate that may occur as a result of performing multiple pairwise comparisons, we used the Bonferroni correction method for adjusting the p-values.

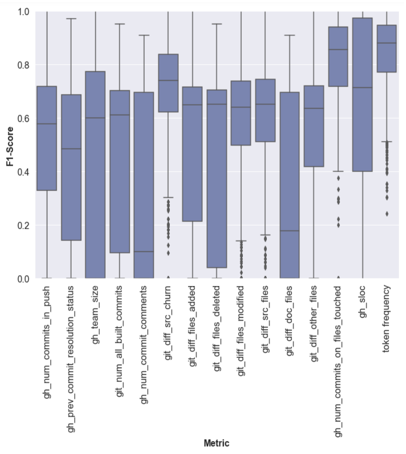
Figures 7.3(a), 7.3(b), 7.3(c) and 7.3(d) are heatmap plots that visualize



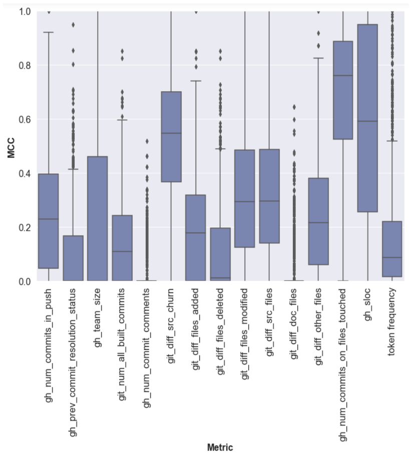
(a) Precision.



(b) Recall.



(c) F1-score.

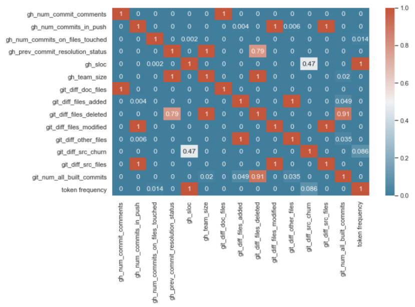


(d) MCC.

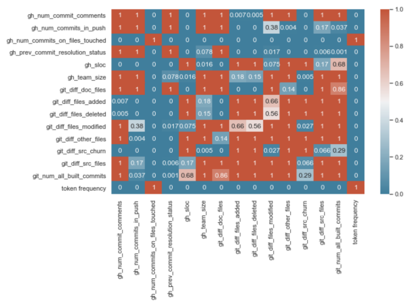
Figure 7.2: The Distribution of the Evaluation Metrics For the TSM and TF metrics Across All Analyzed Projects.

Table 7.5: The Kruskal-Wallis Test Results For Comparing the Performance Values of All Software Metrics.

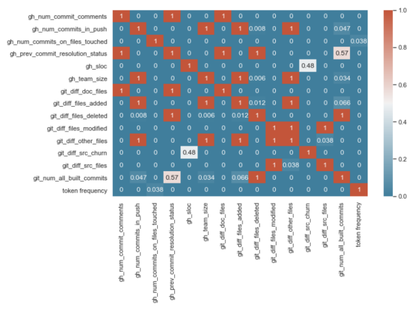
Evaluation Metric	Precision	Recall	F1-score	MCC
Kruska Wallis H	5096.809	442.331	4212.293	6134.276
Sig.	<0.05	<0.05	<0.05	<0.05



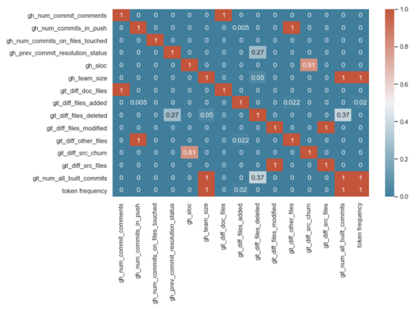
(a) Precision.



(b) Recall.



(c) F1-score.



(d) MCC.

Figure 7.3: Heatmaps showing the distribution of p-values when performing pairwise comparisons between the scores of each evaluation metric for each pair of metrics. Darker cells indicate smaller p-values, and lighter cells indicate larger p-values. Orange cells indicate no statistically significant difference.

the distribution of p-values obtained from each post hoc pairwise comparison between the precision, recall, F1-score and MCC variables. The lower the p-value between each pair of software metrics (<0.05), the more confident we can be that there is a statistically significant difference between them.

By inspecting the p-values in Figures 7.3(a), 7.3(b), 7.3(c) and 7.3(d), we draw the following observations:

- predicting build outcomes using the token frequency or the `gh_num_commits_on_files_touched` metrics results in a statistically significantly different recall and F1 scores than those attained when using each of the other TSM metric (with $p < 0.05$).
- the precision scores attained when using the token frequency metric is significantly different compared to the precision scores attained when using the majority of the other software metrics. The only two exceptions were with the `git_diff_src _churn` and the `gh_sloc` metrics, where no statistically significant difference could be drawn.
- using the `gh_num_commits_on_files_touched` results in a statistically

significant difference with respect to MCC compared with all the other examined metrics (p <0.05).

Based on these observations, the hypothesis that *The mean precision is the same for a model trained on token frequency and each traditional software metrics (H_{0p})* can be rejected except for the `git_diff_src_churn` and the `gh_sloc` metrics, since no significant difference was captured with these. This observation brings us to believe that using the token frequency metric is more effective than twelve of the fourteen other traditional software metrics in identifying passing builds. Similarly, our results reveal that the precision scores recorded when training a model on the `gh_num_commits_ on_files_touched` metric were significantly different than all the other precision scores attained when using each software metric. Thus, we observe that using the count of unique commits on the files included in builds within 3 months of last commit is a more reliable predictor for identifying passing builds, compared to the other metrics.

On the other hand, the hypotheses that *The mean recall and F1-score are the same for a model trained on token frequency and each traditional software metrics (H_{0r} and H_{0f})* can be rejected. This is because a statistically significant difference was captured between the recall scores attained when using token frequency and every other software metric. Hence, using the token frequency metric for training a classifier on predicting build outcomes is more effective for identifying the highest amount of relevant builds that will pass.

Similarly, the hypothesis that *the mean MCC is the same for a model trained on token frequency and each traditional software metrics (H_{0mcc})* can be rejected for all of the other traditional software metrics except for the `gh_team_size` and the `gh_num_all_built _commits`, which were not statistically significantly different with the MCC scores attained by the token frequency metric.

7.4.2.2 Effect Size

Table 7.6 summarizes the effect size results for the TF and `gh_num_commits_on _files_touched` metrics. The `gh_num_ commits_on_files_touched` metric was chosen since it outperformed the other TSMs with respect to precision, recall, F1-score, and MCC. While the calculated p-values in Figures 7.3(a), 7.3(b), 7.3(c), and 7.3(d) indicate that there is a statistically significant difference between the precision, recall, and F1-score produced when using `TF` and `gh_num_commits metrics`, the Cliff's Delta analysis shows that the difference in effect size between the two metrics is relatively small ($<|0.3|$). On the other hand, the effect size between the MCC scores was found to be large, indicating a large difference when using the two metrics respectively ($>|0.8|$).

While the difference in effect size between `TF` and `gh_num_ comments_on_files_touched` is small with respect to precision, recall, and F1-score, the advantage that `TF` has over the TSM is the fact that it operates on a fine-grained level, which allows developers to pinpoint lines of code that require debugging before committing new code changes. A line of code

Table 7.6: The Cliff's Delta Analysis Results Between the Four Evaluation Metrics, Comparing Token Frequency and gh_num_commits_on_files_touched

Name	**Effect size (delta_estimate)**	**Lower**	**Upper**
Precision	-0.23	-0.28	-0.18
Recall	0.12	0.07	0.17
F1	0.09	0.04	0.14
MCC	-0.83	-0.85	-0.80

example from the 'Cloudify' project [5] that was correctly identified by the TF-trained model to trigger a build failure is "`\t\t\t\t\tif (!(Boolean) session.get(Constants.INTERACTIVE_MODE)) {`".

RQ. How effective is the token frequency metric in comparison with traditional software metrics for predicting build outcomes in CI?

Our experiment on 117 Java projects revealed that using the token frequency metric for training a classifier on build outcome prediction yields a slightly better predictive quality for the passed builds than when using the best traditional software metrics. On the other hand, the majority of the examined traditional software metrics were found to yield higher predictive quality than the token frequency metric when it comes to the failed builds.

Overall, the findings of this study suggest that both line and file level metrics are effective for learning build outcome predictions in CI. However, the usage of each metric type may vary depending on the business needs and target domain in which the system operates. For instance, practitioners that would prioritize fast releases over capturing issues in the system might opt to use line-level metric for training as it allows developers to be more confident to start using the code base for implementing new features without waiting for the build to finish running if it was predicted to pass. This assumption needs to be validated in future studies. On the other hand, if practitioners are working on developing safety critical systems, then capturing all issues in the system is prioritized over fast releases. In this case, using the file level metrics is more desirable since those were shown to yield higher effectiveness rate in alerting developers about issues in the code that require fixing.

7.5 Threats to Validity

In this section, we discuss the limitations of our paper using the framework recommended by Wohlin et al. [46].

[5]https://github.com/CloudifySource/cloudify

External Validity: Our study investigates the effectiveness of fifteen different software metrics in predicting build outcomes for 117 Java projects. Hence, we are aware that we can not generalize the conclusions drawn in this study to projects that are written in different programming languages. The results reported in this study may vary if we observe projects that are written in other languages or ones that are linked with different CI services. New studies are needed to validate the effectiveness of the TF metric for projects written in different languages and CI services.

Construct validity: The most tangible threat to construct validity is that we can not assert whether all collected build records with 'failed' status were due to, for example, environmental breakages or faults in the code. Hence, the likelihood of analyzing build records that failed due to factors that are not related to the code-base can not be ruled out. Another threat lies in the validity of the number of build records that we extracted from the TravisTorrent data-set, and whether these records truly mirror the actual builds in Github. However, we minimize these threats by collecting and analyzing a large sample of build records and lines of code from 117 projects.

Internal Validity: A potential internal threat is the presence of undetected flaws in the measurement tools used for extracting both the TSM and the TF. This threat was reduced by carefully inspecting the scripts and testing them on different subsets of data.

Conclusion validity: The main conclusion drawn from this empirical study suggests that using the a line-level metric produces similar predictive quality as file-level metrics in predicting passing builds. This conclusion is based on measuring the effect size of TF and one file-level metric on four predictive performance metrics. Hence, the results might differ if we measure the effect size of TF and another file-level metric, such as gh_sloc. However, we chose the gh_num_comments_on_files_touched metric since it scored highest in mean precision, recall, F1-score, and MCC values, and since it showed a significant difference with almost every metric.

7.6 Conclusion and Future work

In this paper, we conducted an empirical study to examine the effectiveness of a line-level metric in learning build outcome prediction, and compared its effectiveness with fourteen traditional software metrics. We found a a small difference in effect size between token frequency and the best traditional software metrics metric for the passed builds, and a large difference for the failed builds. We conclude that using the line-level metric for training a classifier on build outcome prediction is slightly more effective than the file level metrics for the passed builds, but substantially less effective when it comes to the failed builds. The benefit that token frequency has over traditional software metrics is its ability to pinpoint lines of code that require fixing.

While our analysis revealed promising results regarding the effectiveness of line-level metric for build prediction, future work aims at analyzing additional software metrics such as those related to testing (e.g., gh_tests_ added) and others that operate on line-level. Another future direction is to experiment the use of token frequency on software written in different programming languages. Finally, we would like to evaluate the effectiveness of using both types of metrics - line and file levels - on the predictive quality of a model for build outcome predictions in CI.

Chapter 8

Paper G

The Impact of Class Noise-handling on the Effectiveness of Machine Learning-based Methods for Build Outcome and Code Change Request Predictions

Al-Sabbagh, K.W., Staron, M., and Hebig, R.

Submitted to ACM Transactions on Software Engineering and Methodology.

Abstract

Machine learning-based methods are commonly used to expedite feature delivery to end-users. These methods leverage large amounts of historical code changes to train models on predicting issues in the code-base that can lead to delays in the delivery of features. However, the presence of noise in such data impedes the predictive performance of these methods. In this study, we explore the effectiveness of two statistical-based techniques and a domain knowledge-based technique to handle class noise in predicting build outcomes and code change requests. Our experiments use data from 112 Java open-source projects. The results show that the two statistical-based techniques have a positive impact on the model's performance in both contexts. Specifically, in build outcome prediction, applying the 'majority filter' improves F1-score from 82% to 97%, and MCC from 0.13 to 0.58. Similarly, in code change request predictions, the 'majority filter' improves the F1-score from 17% to 53%, and MCC from -0.03 to 0.57. On the other hand, using the domain-knowledge-based technique was found to impact the prediction of code change requests only. We conclude that applying statistical-based techniques to the training data of code changes is necessary to leverage the prediction of build outcomes and code change requests.

8.1 Introduction

Software engineering companies are under increasing pressure to deliver high-quality software products to the end-users as quickly as possible [159]. To meet these demands, companies are adopting the practice of continuous integration (CI), which promotes for frequent integration and testing of code changes [2]. This practice has become an integral part of modern software development processes as it offers various advantages, such as streamlining workflows, fostering collaboration between team members, and improving software quality.

Despite these benefits that CI offers to companies, it poses the challenge of minimizing the feedback latency between CI and software engineers without compromising fault detection capability. The growing complexity of features and the lengthy compilation time that CI takes underline the need for tools that can expedite the feedback time to software engineers and effectively pinpoint faults in the code. These tools aim to promptly identify and report code changes that are likely to contain faults to software engineers, ensuring rapid detection and fixing of faults.

Machine learning (ML)-based methods have proven effective in accelerating code integration by predicting fault-prone software modules [160][161]. By analyzing code commit histories and build execution outcomes, these methods can identify patterns in code changes that may lead to build failures [162]. However, recent studies have shown that code commits data come with large amounts of inaccurate class values, known as class noise, which can impede the predictive performance of ML models [6]. In the context of CI, these inaccurate values can originate from several factors, such as machinery failure, flaky test cases, data collection methods, etc.

To address the problem of class noise, researchers have proposed several techniques that can be divided into three categories: 1) tolerance, 2) removal, and 3) correction. The tolerance-based techniques focus on designing machine learning models that are robust and capable of tolerating a certain level of noise in the data [54]. The removal-based techniques seek to identify entries with class noise and then remove them from the data-set. The main advantage of using such techniques is that they retain cleaner entries of the data and discard unclean ones. However, using removal-based techniques can lead to losing valuable information that can be used by the model to learn important patterns. Finally, techniques in the correction category seek to correct mislabeled entries by replacing their values with ones that are more appropriate. These techniques are particularly appealing when the training data is small since no removal of data entries is required. However, by correcting mislabeled entries we introduce a risk of bias toward one of the classes [7].

Several SE researchers have investigated the use of different noise-handling techniques to improve the detection of fault-prone software modules. For example, Khoshgoftaar and Rebours [39] conducted a case study to investigate the effectiveness of two removal-based techniques in improving the identification of fault-prone software modules. The evaluation was performed using a data-set of 10.883 software modules. The results showed that the models' performances improve the most when using a conservative technique where fewer entries in the data are removed. In another research study, Hulse et al. evaluated a correction-based technique that uses Bayesian multiple imputation to combat the problem of class noise in faulty software modules [163]. Using a data-set of 282 software program modules, the authors evaluated the error rate of the technique in correcting noisy entries. The results showed that 60% of the mislabeled entries were correctly relabeled. Khoshgoftaar et al. [164] proposed a removal-based technique that uses a statistical model and Boolean rules to improve the prediction of faulty software modules. The evaluation results was done using seven software programs from the NASA data-set. The results showed that the approach was effective in detecting all known noisy entries in 6/7 examined software projects.

From this brief review of research studies, we see that that most studies have used statistical-based models to battle the problem of class noise and evaluated their effectiveness using relatively small samples of software programs. This poses a challenge in the ability to generalize the findings outside their SE contexts. We also observe that little emphasis has been given to investigate and compare the impact of different strategies of class noise-handling in the same SE data-sets. These shortcomings indicate a need to understand the impact of class noise-handling techniques in more SE contexts and using bigger samples of SE data.

These limitations highlight the need for a deeper understanding of the effects of class noise-handling techniques in a wider range of software engineering contexts and with larger data-sets. In light of this need, we set out to examine the effectiveness of three class noise-handling techniques on the performance of a machine learning-based method for predicting fault-prone code changes using large data-sets of code commit histories. The aim is to provide actionable insights to DevOps and software engineers with the most effective class noise-handling technique for building accurate models in predicting fault-prone code changes.

To achieve this goal, we design and implement two computationally controlled experiments wherein we examine the effectiveness of two statistical-based techniques and a domain knowledge based technique in the context of CI. In the first experiment, we examine the impact of applying each of the three techniques to a training data of code commits and build job execution outcomes for build outcome prediction. We perform the evaluation using a large data-set of 110 Java Open-source projects. The goal is to examine whether applying any of the three class noise-handling techniques improves the prediction of build job outcomes.

RQ1: What is the impact of applying class noise-handling techniques on predicting the outcome of builds in continuous integration?

The second experiment examines how each class noise-handling technique affects the performance of an ML-based method in predicting code change requests during code review. We select a sample of 5066 lines of code and corresponding code comments from two open-source projects for evaluation. These code commits are manually labeled based on the sentiment expressed by reviewers regarding the readiness of the code for integration or the need for changes. Consequently, we pose the following research question:

RQ2: What is the impact of applying class noise-handling techniques on predicting code change requests?

We evaluate the impact of the three class noise-handling techniques in the two experiments by measuring the predictive performance of a random forest (RF) model in terms of Precision, Recall, F1, and MCC. We evaluate the effectiveness of each technique by comparing the four performance metrics before and after applying each technique respectively.

Our results from the first experiment suggest that applying the removal-based techniques would consistently improve the predictive performance of the model in terms of Precision, Recall, F1, and MCC. In addition, the results from the second experiment suggest that applying any of the three class noise-handling techniques will consistently improve the the predictive performance of the model, albeit to a varying degree.

The remainder of this paper is structured as follows: Section 2 discusses previous research studies that are related to the study presented in this paper. Section 3 presents background information, covering core concepts and a formal definition of how class noise is measured in code commits data. Section 4 describes the research design. Section 5 presents the evaluation results of the impact of class noise-handling techniques on the performance of the ML-based method for build outcome and negative review comment predictions. Section 6 answers the research questions and discusses general observations. Section 7 addresses the threats to validity of this study. Finally, Section 8 presents the findings and concludes the paper.

8.2 Related Work

The literature provides several techniques for handling class noise, including removal, correction, and tolerance-based ones. In our research, we specifically focus on removal and correction-based techniques as they pertain to our analysis. Therefore, we highlight some of the most popular and widely used techniques that fall under these two categories.

8.2.1 Removal and correction based techniques for class noise-handling

Removal based techniques Brodley and Friedl [21] proposed a method for identifying and handling noise in training data using three or five learning algorithms as filters. These filters employ majority voting or consensus filter mechanisms to identify potentially noisy instances. The effectiveness of their approach was assessed using five different data-sets. The evaluation results indicated that, when dealing with noise levels of 20% and higher, the majority filter exhibited slightly better prediction accuracy compared to the consensus filter. However, for noise levels below 20%, both filters demonstrated similar accuracy scores ranging from 79% to 82%.

Guan et al. [14] proposed a variant of the majority and consensus filters that incorporated a semi-supervised classification step to aid in predicting class values for unlabeled instances. The technique was tested on 20 benchmark data-sets and evaluated by measuring the classification error rate. The comparisons demonstrated that the technique improved the classification performance of machine learning models across four different noise ratios. Specifically, when the noise ratio was 10%, the technique reduced the classification error rate by 4.5%, while at a noise ratio of 40%, the improvement significantly increased to 25.6%.

Wilson and Martinez [165] introduced an instance-based technique called DROP3, which aims to remove noisy entries from the data. This technique employs a distance function to determine the proximity of each input vector in a subset of the training data to the entire data-set. To evaluate its effectiveness, DROP3 was compared with ten other techniques across 31 classification tasks. The comparison results based on average accuracy demonstrated that DROP3 ranked 5th in terms of its improvement effect on accuracy, with an average of 81.14%.

Correction based techniques Muhlenbach et al. [40] introduced a algorithm that allows users to filter and polish noisy instances. The algorithm utilizes neighbourhood graphs and cut edge weight algorithms to identify noisy suspects in the data. An instance is considered noisy when its class value is different than one of the examples belonging to its geometrical neighbourhood. Once noisy suspects are identified, they can be either removed or removed and relabelled. Muhlenbach et al. evaluated their algorithm using data-sets extracted from 10 benchmark ML repositories. The findings suggest that removing noisy suspects from the training data produces better results in 9 out of 10 data-sets compared to removing and relabelling noisy suspects at a noise level between 4% and 10%.

Teng [20] introduced a noise correction technique that exploits the interdependence relationship between the attribute and class values to identify and correct inaccurate values. The idea is to use an ensemble classifier to predict noisy suspects in the data. Some entries that were incorrectly classified by the ensemble (using a voting scheme) are tagged as noisy and then corrected. The author evaluated the effectiveness of her technique by measuring the prediction accuracy of Decision Trees before and after correction was applied. The results showed that at an intermediate noise level (20–30%), the improvements in accuracy were significant. However, at 40% noise level, the improvements in accuracy were inconsistent.

In comparison with the correction-based noise-handling techniques highlighted above, the domain knowledge-based technique that we use in this study is lightweight and less complex since it relies on our knowledge in the domain of code changes. An earlier investigation on the effectiveness of the technique was presented in our recent work [98]. There, we demonstrated that applying the technique to industrial data of code changes and test execution outcomes

Table 8.1: Results summary for class noise experimentation in software engineering research

Study	SE context	Data-sets	Noise approach(s)	Results
Kim et al. [166]	Defects Prediction	- Columba, - Eclipse 3.4, - Scarab, - SWT, - Debug	- Removal	F1= 71%
Liebchen et al. [167]	Effort Estimation	- EDS	- Removal - Correction	% of remaining noise: Removal: 11.24% Correction: 365%
Zhong et al. [168]	Defects Prediction	- NASA: JM1 and KC2	- Removal	Noise Recall performance: 77% in JM1 - 91% in KC2
Seiffert et al. [169]	Defects Prediction	- CCCS	- Tolerance	AUC: ranged from 97% to 100%

improves the predictive performance of a model for test case selection from 44% to 81% Precision and 17% to 87% Recall. In this study, we extend the evaluation of the technique by comparing its effectiveness with two statistical-based models in improving the prediction of build job outcomes and code change requests.

8.2.2 Class noise-handling in software engineering contexts

The most related studies to our work concern the analysis of the impact of class noise-handling on the predictive performance of ML models for solving SE tasks. This section highlights research studies that examine the effect of class noise-handling techniques on the predictive performance of ML models in SE contexts.

Table 8.1 summarizes previous research studies that examined the effect of class noise-handling techniques on the predictive performance of ML models in SE contexts.

Kim et al. [166] proposed the Closest List Noise Identification technique for removing mislabelled entries in software defect data sets. The technique uses Euclidean Distance for measuring similarities between entries in the training data and comparing their class values. If the percentage of entries that have different class values, relative to an entry, is above a predefined threshold, then that entry is treated as noisy and, thus, removed from the data-set. The technique was evaluated on data-sets from the Eclipse 3.4 SWT and Debug open-source projects. The prediction results attained by an SVM model showed that F1 improves from 34% to 71% when the noise level is exactly at 30%. On the other hand, F1 decreased for the same SVM model when the noise level was higher than 30% (F1 decreased from 35% to 24%).

Liebchen et al. [167] conducted an experiment to assess the performance of three class noise-handling techniques using a sample of 8888 entries. The sample data describes completed software projects whose attributes capture projects' characteristics (e.g., project names and types). The examined class noise-handling techniques were 1) filtering, 2) robust filtering, and 3) filtering and polishing. The results of the experiment showed that using a robust filtering technique could eliminate 88% of noisy entries found in the initial data-set. Conversely, the filtering and polishing technique resulted in tripling the amount of noisy entries compared to the amount of noise found in the initial data.

Zhong et al. [168] proposed using an unsupervised learning technique followed by manual labeling by experts to deal with the problem of class noise and missing class labels in software-fault measurement data. The idea is based on the assumption that fault-prone software modules will have similar software measurements and, thus, can likely form clusters. The authors classified their technique as a removal-based approach since it offers experts the ability to decide whether all grouped fault-prone software modules should be labelled as such or not, and accordingly decide whether to keep or remove them from the data. The evaluation of the technique was done by comparing the mislabeled software modules by SE experts with software modules tagged by another removal-based technique, described in [24]. The evaluation was done using two software projects (written in C++) from the NASA software projects. The results showed that their clustering technique achieved a noise Recall performance of 77% in the first project and 91% in the second project.

Seiffert et al. [169] conducted a series of experiments to investigate the impact of both class noise and class imbalance on the predictive performance of models built to predict fault-prone program modules. The authors investigated the robustness of 11 ML models in tolerating class noise when using 7 data-sampling techniques. By seeding class noise into a data-set that contains 282 program modules written in Ada, the authors examined the impact of applying the 7 sampling techniques in the presence of four different class noise levels on the performance of the 11 models. The average Area under the ROC Curve (AUC) showed that the Naive Bayes model performs better than all others at all noise levels and is relatively unaffected by the increase in noise ratio. Specifically, the average AUC of the Naive Bayes model ranged from 97% to 100% at the four seeded noise levels.

From this brief overview, we observe that the evaluation of the techniques was performed using small sample data-sets (one or a few software programs) and mainly focused on improving defects prediction. Therefore, the study presented in this paper adds to the literature by examining the effectiveness of three techniques using: 1) large real-world data-sets of historical code commits in two unexplored SE contexts (i.e., build outcome and code change requests), and 2) a reliable performance measure – MCC – which reveals more truth about the bias introduced by the noise-handling techniques.

8.3 Background

This section provides the definition of class noise that we used in this study. It also explains how the code review process in Gerrit is carried out, and describes the three class noise-handling techniques that we examine for effectiveness in this study.

8.3.1 Definition and example of class noise

In this study, we define class noise as *the ratio of contradictory entries in each class to the total number of entries in the data.* `contradictory entries` are entries that have the same vector representation and are labeled with different class values. Based on this definition, we use the following formula to measure the ratio of class noise in the data:

$$\text{Noise ratio} = \frac{\text{Number of Contradictory Entries}}{\text{Total Number of Entries}}$$

Since a contradictory entry can only be among two or more entries, thus their total count in the data is calculated by summing up the number of the same entries that appear with one or more different class value(s). For example, a data-set containing six of the same entries with five that are labeled with `True` and one labeled that is labeled with `False` has six contradictory entries. It is not possible to define a general rule to identify which class label is correct based on the number of entries [167]. For example, noise sources might systematically tend to introduce False "False" class values. Since we do not know exactly which class value should be used in a specific context, we cannot simply re-label any entry, as suggested by the currently used solutions (e.g. using entropy measurements [52]), and therefore we count all such entries as contradictory.

8.3.2 Code review process in Gerrit

Over the last few decades, the process of code review has started to become more prevalent in the software engineering community. Big companies like Google started adopting lightweight tools to accelerate the review process [170]. A reviewer process typically enacts four sequential steps: 1) the author of a commit submits it to the code review tool, 2) the reviewers review the changes introduced in the submitted commit, 3) the reviewers can then either initiate a discussion thread on specific lines/blocks in the committed code or approve the changes, 4) the author of the commit either responds by addressing the comments raised by the reviewers or arguing against the proposed changes. This feedback loop between team members remains active until the majority of the reviewers are satisfied with the discussion/modifications or until the submitted commit gets discarded.

Gerrit is an example of such tools that have recently gained popularity in OSS projects and several other Google projects (e.g., Chromioum). In Gerrit, any submitted commit is only merged into the master branch if the assigned reviewers and the automatic checker have explicitly approved the changes in the submitted commits. Gerrit utilizes a voting mechanism that enables code reviewers to indicate their level of approval on merging new code changes. The voting mechanism uses a scale of intervals that span between -2 and +2. An interval of:

- +2 indicates that the change looks good and is approved.
- +1 indicates that the change looks good, but someone else must approve it.
- 0 indicates no score.
- -1 indicates that it is not preferable to submit this commit.
- -2 indicates that the change is rejected.

This voting mechanism, together with the code review discussions, constitutes the evidence needed by team members to either integrate or rework the proposed change.

8.3.3 Class noise example in code review data

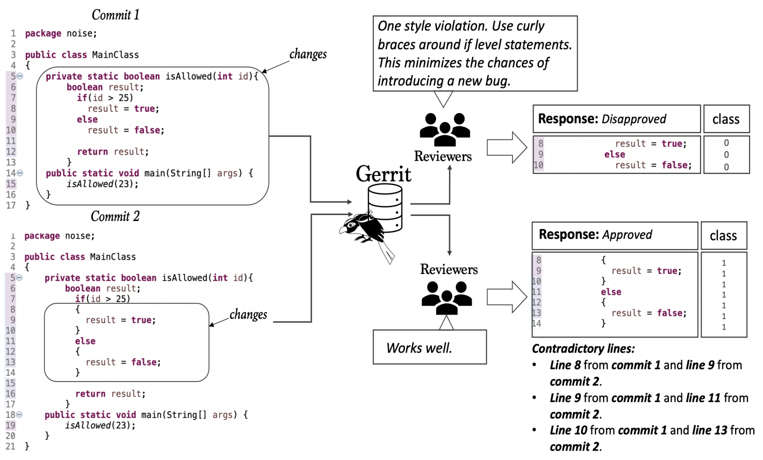
Figure 8.1 exemplifies a scenario in which class noise can be introduced into a code review data-set. The Figure shows two code commits written in Java – commit 1 and commit 2 – submitted for peer review inspection via Gerrit. The assigned reviewers to commit 1 agree to decline the merge request – the majority of reviewers voted with a score below 0 – and start a discussion thread with the author, requesting a fix to the code. On the other hand, the assigned reviewers to commit 2 agree to accept the commit – the majority of reviewers voted with a score above 0 – and, hence, merge it into the master branch.

If we use the sentiment of reviewers to label each line of code in the two commits, then every line of code in commit 1 will be labeled with 'disapproved' and every line of code in commit 2 will be labeled as 'appproved'. Note that in this example, we use a class value of '0' to annotate a line of code that is disapproved and a class value of '1' to annotate a line of code that is approved. Accordingly, the following pairs of lines from commits 1 and 2 can be classified as contradictory, since they are duplicates and hold different class labels:

- line 8 from commit 1 and line 9 from commit 2.
- line 9 from commit 1 and line 11 from commit 2.
- line 10 from commit 1 and line 13 from commit 2.

Since there are a total of 19 lines of code in the changes of commits 1 and 2, then the total number of entries in the training data is 19. The formula for calculating the noise ratio for this example is thus:

$$\text{Noise ratio} = \frac{6}{19} = 31.5\%$$



Commit 1

```
1  package noise;
2
3  public class MainClass
4  {
5      private static boolean isAllowed(int id){
6          boolean result;
7            if(id > 25)
8              result = true;
9            else
10             result = false;
11
12           return result;
13         }
14     public static void main(String[] args) {
15         isAllowed(23);
16     }
17 }
```

Commit 2

```
1  package noise;
2
3  public class MainClass
4  {
5      private static boolean isAllowed(int id){
6          boolean result;
7            if(id > 25)
8            {
9              result = true;
10           }
11           else
12           {
13             result = false;
14           }
15
16           return result;
17         }
18     public static void main(String[] args) {
19         isAllowed(23);
20     }
21 }
```

One style violation. Use curly braces around if level statements. This minimizes the chances of introducing a new bug.

Response: *Disapproved*	class
`8          result = true;`	0
`9        else`	0
`10         result = false;`	0

Works well.

Response: *Approved*	class
`8        {`	1
`9          result = true;`	1
`10       }`	1
`11       else`	1
`12       {`	1
`13         result = false;`	1
`14       }`	1

Contradictory lines:

- ***Line 8*** from ***commit 1*** and ***line 9*** from ***commit 2***.
- ***Line 9*** from ***commit 1*** and ***line 11*** from ***commit 2***.
- ***Line 10*** from ***commit 1*** and ***line 13*** from ***commit 2***.

Figure 8.1: Class noise in code review data.

8.3.4 Noise-handling techniques

In our work, we examine the effectiveness of two removal-based techniques and a domain-knowledge-based class noise-handling technique. We begin this section by describing the two removal-based techniques and then turn to describe the third technique.

8.3.4.1 Removal-based noise-handling techniques

From a wide range of existing class noise-handling techniques, we chose to examine two of the most widely used and reported in the literature. Namely the consensus (CF) and majority filter (MF) introduced by Brodley et al. [21]. The two techniques employ an ensemble of ML models for classifying noisy entries in the training data using a voting mechanism: a univariate decision tree (C4.5), a K-Nearst Neighbors (KNN), and a linear regression model (LR). Figure 8.2 illustrates the main procedure of the techniques for identifying and removing noisy entries. The techniques work in a k-fold cross-validation manner, where for k repetitions k-1 folds are used for training each model in the ensemble. Each model is then used to tag each entry in the remaining hold-out fold as either noisy or clean. At the end of the k repetitions, each entry in the entire data-set is tagged with a label that denotes whether the entry is noisy or not. Finally, a decision about which entries should be treated as noisy – and thus removed – or not is made using a voting mechanism and accordingly. It is worth noting that CF is considered more aggressive than MF, since it removes a higher proportion of entries from the data [24].

Based on the majority voting mechanism described in [21], an entry that gets tagged as noisy by more than 50% of the models must be treated as such and thus removed from the data. On the other hand, the consensus filter voting mechanism follows a more conservative approach suggesting that if an entry gets tagged by one or more models as noisy then it should be treated as such

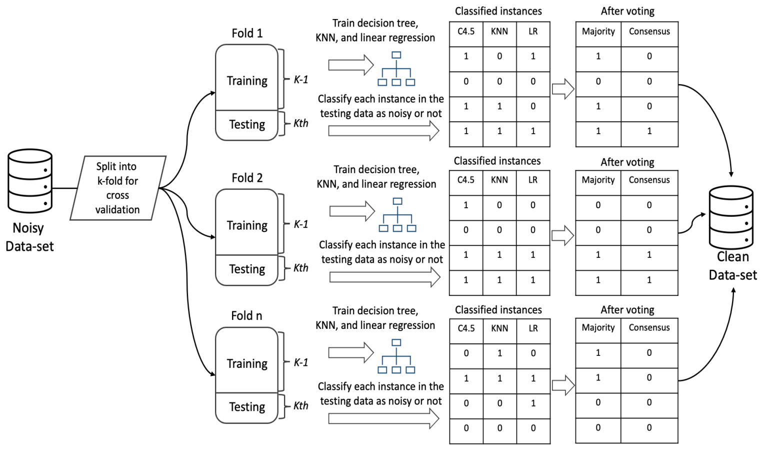

Noisy Data-set
Split into k-fold for cross validation
Fold 1
Fold 2
Fold n
Training
Testing
K-1
Kth
Train decision tree, KNN, and linear regression
Classify each instance in the testing data as noisy or not
Classified instances
C4.5
KNN
LR
After voting
Majority
Consensus
Clean Data-set

Figure 8.2: noise-handling procedure using the statistical-based techniques.

and removed from the data-set.

8.3.4.2 Domain knowledge-based noise-handling technique (DB)

We complement the analysis by examining the effectiveness of another technique – termed domain knowledge base (DB) – that we developed and published in a previous study [98]. Unlike statistical-based techniques (such as CF and MF), the DB was designed based on our knowledge in the domain of source code changes to correct and remove noisy entries. The procedure of the technique can be summarized by the following steps:

- sequentially assigns a unique 8-digit hash value for each line of code in the original data set.
- creates an empty dictionary for storing unfiltered entries.
- iterate through the set of hashed entries in the original data set and save syntactically unique entries into the dictionary.
- compare the class values between each pair of duplicate entries, both in the original and dictionary sets. If the two values are different and the value of the entry in the original set is annotated with the 'positive' class, then relabel the entry in the dictionary from 'negative' to 'positive' and discard the entry in the original set. If both entries have the same class values, then add the entry from the original set to the dictionary.

This way of handling class noise can be seen as both corrective and removal, since it 1) corrects the class value of the same entries that first appear in the 'negative' class and then the 'positive class', and 2) removes one entry in each pair of contradictory entries.

In the context of code reviews and build predictions, problematic lines of code often occupy a small proportion of the overall code fragment of commits.

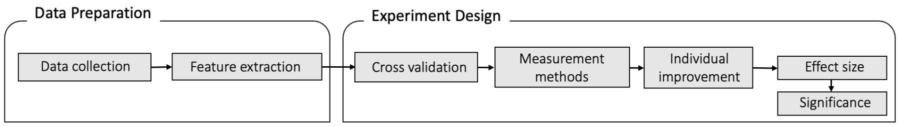


Figure 8.3: Illustration of the research design activities.

Thus, a code fragment that was negatively perceived by a reviewer (i.e., needs to be fixed) is not likely to have issues in every line of code. Similarly, a line of code that appears as part of a passing build is not likely to trigger failure in a build job. Hence, the DB technique ensures to relabel contradictory lines of code from 'disapproved' to 'approved' in code reviews data and from 'failing' to 'passing' in build jobs data – if those lines have already been seen as part of approved reviewed fragments or passing builds.

8.4 Research Design

In the following, we describe the preparation, design, and operations that we carried out for answering RQ1 and RQ2. The first section (8.4.1) describes the independent and dependent variables and how we collected the sample data for answering the research questions. The second section (8.4.2) discusses the design and operations that we carried out.

Figure 8.3 presents the procedure that we performed in each experiment and the main sequence of activities. What follows is a detailed description of each activity presented in Figure 8.3.

8.4.1 Data collection and preparation (Part one)

We begin this section by describing the independent and dependent variables of this study. Then, we describe the data collection and the feature extraction methods that we used to prepare our data.

8.4.1.1 Experiment variables

Independent variables noise-handling technique is the only independent variable (treatment) examined for an impact on the performance of an ML model. Three variations (treatment levels) of the independent variable is examined, namely the CF, MF, and DB techniques. The CF and MF techniques rely on statistics to detect noisy entries, whereas the DB technique relies on domain knowledge to relabel contradictory entries. Note that each treatment level is independent from one another and no cross-level effects are possible.

Dependent variables The dependent variables in this study are four state-of-the-art metrics that we use to evaluate the impact of each treatment level in experiments 1 and 2. The four metrics are Precision, Recall, F1, and MCC.

In the context of build outcome prediction, Precision expresses the proportion of correctly predicted lines of code that do not trigger build failures to the total number of lines predicted as such. A high Precision means that the model performs well in predicting lines that do not trigger build failures. Recall expresses the proportion of correctly predicted lines that do not trigger build failures to the total number of lines that actually do not trigger build failures. A high Recall indicates that many of the passing builds are correctly recognized by the model.

In the context of code change request prediction, Precision expresses the proportion of correctly predicted lines of code that do not contain quality issues to the total number of lines predicted as such. Recall expresses the proportion of correctly predicted lines of code to the total number of lines in the program that actually do not contain quality issues.

The F1-score is considered as one of the most popular metrics to evaluate the classification performance of ML models [171]. It uses three elements in the confusion matrix (True positives, False positives, and False negatives) to provide a harmonic mean between Precision and Recall. However, F1-score can sometimes lead to misleading conclusions, as it does not take the fourth element (True negatives) in the confusion matrix into account [171]. Therefore, we decided to complement the analyses of the effectiveness of the three noise-handling techniques by measuring the MCC. A high MCC indicates that the performance of the model is good in predicting the binary classes. Thus, MCC considers what share of the elements in the negative class is correctly identified as negative.

8.4.1.2 Collection of build outcomes data (Experiment 1)

The study subjects used in the first experiment were extracted from a public repository that we published in a previous work [162] The repository comprised a total of 49,040 build records that belong to 117 Java projects and their corresponding code change commits. Each record holds information about the execution outcome of a build job (passed/failed) executed against a commit. The repository also contains all added/modified code changes that were built by the CI server used in each project. In addition, the repository includes a set of feature vectors that represent the extracted code changes using the token frequency metric described in [162].

In the empirical study presented in this paper, we began the analysis by using build information and historical code changes of all 117 Java projects. However, when applying the DB technique to the code change data we observed that the distribution of the binary classes in seven projects reached a class ratio close to [0% to 100%]. Since using such imbalanced data for training would lead to creating a model that either makes pessimistic or optimistic decisions (i.e., always considering any entry as belonging to one of the classes), we decided to exclude these projects from the analysis and work on examining

Table 8.2: Excluded projects from the analysis due to class imbalance after applying the DB technique

Project	class 0	class 1	class ratio
azkaban	6817	58539	11.64%
intellij-elixir	985	242697	0.4%
java-design-patterns	55	23249	0.24%
jinjava	8	12747	0.06%
jsonschema2pojo	3	15091	0.02%
nodeclipse-1	27	23959	0.11%
picard	378	11017	3.43%

the impact of the class noise-handling techniques in the remaining 110 projects. Table 8.2 summarizes the distribution of classes in the seven excluded projects.

The distribution of classes and their ratios in each analyzed project can be found in Table 8.20 in Appendix 8.9. The original and curated versions of the dataset, which contain the data after applying the noise-handling techniques, are available on Zenodo [1].

8.4.1.3 Collection of code reviews data (Experiment 2)

The results presented in this paper for answering RQ2 are based on two Java open-source projects, namely 'Wireshark' and 'Google sources'. Our data collection process began by searching for publicly accessible projects that utilize Gerrit as their code review platform. We chose Gerrit for two main reasons. Firstly, it provides a REST API that facilitates the development of a mining tool for code review comments. Secondly, it provides a web interface that allows for tool validation.

In total, we collected code commit and code review data from 16 projects, obtained from two different sources. The first source was a public repository where we collected a total of 144,706 code review comments and their corresponding code changes from 15 Java-based projects [172]. This repository contains a set of 'json' files that provide mapping information, linking review comments to specific locations within patch files where the reviewed lines of code are found. By accessing this mapping information, we developed a tool for mapping code review comments and committed code changes into the same files. A summary of the project names and the number of extracted lines of code from the repository can be found in Table 8.3.

The second source of data is a Gerrit repository that hosts a project called 'Google Sources' [2]. To extract code review comments and their corresponding code changes, we developed a another tool for mining code commits and review comments from this Gerrit repository.

The design of the tool focused on extracting data only from code that was either modified or added, excluding deleted code, as predicting issues in deleted

[1] https://doi.org/10.5281/zenodo.8023970

[2] 'https://gerrit-review.googlesource.com'

Table 8.3: The study subjects extracted from the first data source.

Id	Project	Number of lines of code and their review comments
1	acumos	1305
2	android	19977
3	asterix	22305
4	carbonrom	28
5	cloudera	7855
6	eclipse	16602
7	fd.io	830
8	gerrithub	1978
9	googlereview	22837
10	iotivity	1244
11	omnirom	358
12	opencord	89
13	polarsys	371
14	unicorn	25
15	wireshark	48902

changes falls outside the scope of this study. Following the approach of previous studies [101], [98], and [173], we were particularly interested in examining the impact of class noise-handling techniques on modified and added lines of code. The implementation procedure of the code mining tool can be summarized as follows:

- The tool sends an API request to retrieve all review, change, and file IDs.
- For each change and revision ID, it requests the IDs of all reviewed files and their corresponding review comments.
- For each reviewed file, the tool sends a get request to retrieve all code changes that were either added or modified.
- The retrieved code changes in a reviewed file are then mapped with the review comment made by the reviewer.

Applying this tool to the 'Google Sources' repository resulted in the collection of a total of 74,003 lines of code and their associated review comments. To ensure the accuracy of our tool, we cross-verified the review comments of two file IDs in a revision with those manually extracted using the Gerrit web interface.

Labeling code review comments After collecting the data-sets, we decided to manually label a sample of the collected code comments from each projects. The labeling guideline was as follows:

- If a code comment explicitly requested a change to the code, it was labeled as '0'.

- If a code comment accepted the change or expressed praise for it, it was labeled as '1'.
- If a code comment suggested an optional change or raised a question, it was labeled as 'neutral'.

To ensure inter-annotator agreement and to refine the guideline, a pilot annotation was conducted. In this step, the three authors of this paper independently annotated a sample of 200 code comments. The sample was randomly selected and included a mix of comments that requested a change and ones that did not. The comparison between the annotations revealed a 95% agreement for comments requesting a change and an 80% agreement for comments approving the code change for integration. Accordingly, we considered this rate of inter-annotator agreement to be sufficient and decided to proceed with annotating a larger sample of the data.

Table 8.4: The distribution of the annotated comments with respect to the '0', '1', and 'neutral' classes

Id	Project	class_0	class_1	class_neutral
1	unicorn	16	1	8
2	wireshark	267	110	67
3	googlesources	166	66	62
4	asterix	341	5	142
5	googlereview	276	0	96
6	acumos	405	8	57
7	iotivity	1001	2	131
8	fd.io	284	9	95
9	android	329	12	152
10	carbonrom	13	2	14
11	opencord	53	0	33
12	omnirom	131	0	41
13	gerrithub	291	9	94
14	eclipse	504	84	188
15	polarsys	167	0	28
16	cloudera	377	5	113

Subsequently, the three authors collectively annotated a total of 6,255 code comments, with each author annotating approximately a third of the comments. After completing the annotations, an analysis of the distribution of labeled comments was performed. It was observed that, on average, 95% of the annotated comments fell into the '0' and 'neutral' classes as summarized in Table 8.4.

Considering the imbalanced nature of the data-sets, we decided to annotate a larger sample of comments. After reviewing all code comments within the 16 collected projects, we found that only two projects, namely Wireshark and Google sources, contained a fairly balanced distribution of comments requesting

Table 8.5: The distribution of annotated code comments.

Project	Class 0	Class 1
Wireshark	1665	897
Google sources	1552	894

a code change and accepting a change. Therefore, we decided to focus on these two projects and annotate a larger sample of comments from these projects. A total of 2,562 code comments were annotated in the Wireshark project, while 2,446 code comments were annotated in the Google Sources project. The distribution of the annotated comments is summarized in Table 8.5.

8.4.1.4 Feature extraction

In this study, we employ a textual analysis technique that extracts features from the extracted set of code changes. Each feature corresponds to a code token that appears in the extracted code. In this study, we utilize a tool proposed by Ochodek et al. [42] to perform the features extraction using the bag of words (BoW) model. The textual analysis tool follows the below procedure to extract features:

- creates a vocabulary for all lines of code (using the BoW technique, with a cut-off parameter of how many words should be included[3])
- creates a token for words that fall outside of the frequency defined by the cut-off parameter of the bag of words
- finds a set of predefined keywords in each line,
- checks each word in the line to decide if it should be tokenized or if it is a predefined feature.

The output of this step is a large array of numbers, each representing the the token frequency of a specific feature in the bag of words space of vectors. In this study, we chose to use a bi-gram model for representing the feature vectors, as it was previously shown to yield good learning performance in a similar context (e.g., [98]). Our experimental raw and feature vector data are accessible at [4].

8.4.2 Design of the experiments

In this section, we discuss the design of the two experiments that we carried out to answer RQ1 and RQ2. To ensure uniformity and to control as many confounding factors, we use the same variables and activities in the design and execution of both experiments. The following five activities are carried out:

[3]BoW is essentially a sequence of tokens, which are descendingly ordered according to frequency. This cut-off parameters controls how many of the most frequently used words are included as features – e.g. 10 means that the 10 most frequently used words become features and the rest are ignored.

[4]https://doi.org/10.5281/zenodo.7527590

1. cross validation.
2. measurement method.
3. analysis of individual improvement.
4. analysis of effect size.
5. analysis of significance testing.

8.4.2.1 Cross validation

We applied a 10-fold stratified cross validation partitioning scheme on each data-set in both experiments, i.e., ten random partitions of each project data-set with a combination of nine of them (90%) as training set and the remaining one as a test set (10%). A total of 1,100 training and testing trials were carried out in experiment 1, and 160 training and testing trials in experiment 2. The distribution of the classes in each training fold is then evaluated to decide whether subsequent balancing of the classes for each training fold is required. If the distribution of one class exceeds 60%, then the minority class is over-sampled until all data points in the minority class even out with the number of entries in the majority class. We used the generated testing folds from the original data to evaluate the performance of the ML model before and after applying each treatment level on the training folds respectively.

8.4.2.2 Measurement method

To examine the impact of the three class noise-handling techniques, we conducted an evaluation using a sample of twelve data-sets and three classifiers: Random Forest (RF), Extreme Gradient Boosting (XGBoost), and a two-dense Neural Network (NN). The goal of this evaluation was to identify a suitable classifier for measuring the impact of the class noise-handling techniques.

We selected these classifiers based on their unique characteristics and strengths in capturing patterns in data. RF was chosen for its robustness in predicting similar SE tasks, such as test case selection [98]. XGBoost is an optimized implementation of gradient boosting, and it is widely recognized for its high predictive performance and scalability in different applications, such as [174]. Additionally, neural networks are renowned for their ability to learn intricate representations within the data [175]. The RF and XGBoost models were kept at their default parameter values, as provided by the scikit-learn library (version 0.20.4) [76]. The NN model was a sequential model, consisting of two dense layers, and was implemented using the Keras library [77].

During the evaluation, each of the three models was trained on both the experimental data (exposed to the treatment) and the control data (before applying the treatment). The evaluation results, presented in Table 8.15 in Appendix A, revealed that RF outperformed the other two models when trained on the CF and MF experimental data, as well as when trained on the control data. Consequently, we decided to select RF as the primary method for measuring the impact of the three class noise-handling techniques on the predictive performance of a model for build outcome and code change request predictions.

8.4.2.3 Individual improvement

To understand the impact of each class noise-handling technique on the predictive performance of a model for build outcome and negative review comment predictions, we examine improvement trends in the four dependent variables before and after applying the treatment on each study subject. Another important measurement that we make is the ratio of class noise, as defined in Section 8.3.1, before and after applying the treatment to the training data. The individual impact of each treatment level on the four dependent variables and class noise ratio is visually analyzed using scatter plots.

8.4.2.4 Effect size

To summarize the effect of the three noise-handling techniques on the predictive performance of the RF model in each experiment, we calculate the descriptive statistics of the four dependent variables before and after applying the treatment to the training data. The effect of each technique is visualized by plotting the mean scores and distributions of the four dependent variables.

8.4.2.5 Significance testing

We hypothesize that using any of the three class noise-handling techniques, described in Section 8.3.1, would improve the performance of a model for predicting code change requests and build outcomes, compared to when leaving noisy entries in the training data. Accordingly, twelve hypotheses are formally defined and tested for statistical significance in each experiment. Table 8.6 lists and defines the hypotheses.

All of the hypotheses listed in Table 8.6 are two-tailed, since we are mainly interested in understanding whether the value of each dependent variable would be impacted by any of the three class noise-handling techniques. Each hypothesis is defined in terms of one variation of the independent and dependent variables. For example, the first hypothesis under column 'Precision' suggests that applying MF (a variation of the independent variable) on the training data will result in a significantly different Precision (a dependent variable) score compared to when leaving noisy entries in the training data.

Normality test To determine whether to use parametric or non-parametric statistical tests, we checked for the normality assumption of the four dependent variables using the Shapiro-Wilk test available in the scikit learn library [76]. The results showed that the distribution of the four dependent variables was not normally distributed. Based on the normality test results, we decided to run the Kruskal-Wallis (a non-parametric test) for comparing the Precision, Recall, F1-score, and MCC values between the different treatment levels. The Mann–Whitney U test was then run to perform a pairwise comparison between

Table 8.6: The hypotheses for the effects of noise handling techniques on build outcome and code change request predictions.

Precision	**Recall**	**F1**	**MCC**
H_{0p_m}: The mean Precision is the same for a model trained on MF-curated and non-curated data. $\mu_{0p\_m} = \mu_{1p\_m}$	H_{0r_m}: The mean Recall is the same for a model trained on MF-curated and non-curated data . $\mu_{0r\_m} = \mu_{1r\_m}$	H_{0f_m}: The mean F1 is the same for a model trained on MF-curated and non-curated data. $\mu_{0f\_m} = \mu_{1f\_m}$	H_{0m_m}: The mean MCC is the same for a model trained on MF-curated and non-curated data. $\mu_{0m\_m} = \mu_{1m\_m}$
H_{0p_c}: The mean Precision is the same for a model trained on MF-curated and non-curated data. $\mu_{0p\_c} = \mu_{1p\_c}$	H_{0r_c}: The mean Recall is the same for a model trained on MF-curated and non-curated data . $\mu_{0r\_c} = \mu_{1r\_c}$	H_{0f_c}: The mean F1 is the same for a model trained on MF-curated and non-curated data. $\mu_{0f\_c} = \mu_{1f\_c}$	H_{0m_c}: The mean MCC is the same for a model trained on MF-curated and non-curated data. $\mu_{0m\_c} = \mu_{1m\_c}$
H_{0p_d}: The mean Precision is the same for a model trained on MF-curated and non-curated data. $\mu_{0p\_d} = \mu_{1p\_d}$	H_{0r_c}: The mean Recall is the same for a model trained on MF-curated and non-curated data . $\mu_{0r\_d} = \mu_{1r\_d}$	H_{0f_d}: The mean F1 is the same for a model trained on MF-curated and non-curated data. $\mu_{0f\_d} = \mu_{1f\_d}$	H_{0m_d}: The mean MCC is the same for a model trained on MF-curated and non-curated data. $\mu_{0m\_d} = \mu_{1m\_d}$

the dependent variables under each treatment level and the same measures that we recorded after training on the original data-set.

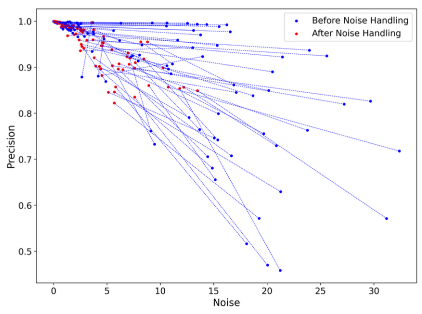
8.5 Results

This section reports the results of the two experiments to answer the two research questions.

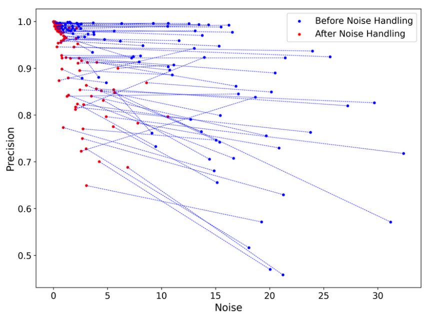
8.5.1 The impact of class noise-handling on predicting the outcome of builds in continuous integration (RQ1)

To address our research question of *What is the impact of applying class noise-handling on predicting the outcome of builds in continuous integration?*, we structure our results into three analyses:

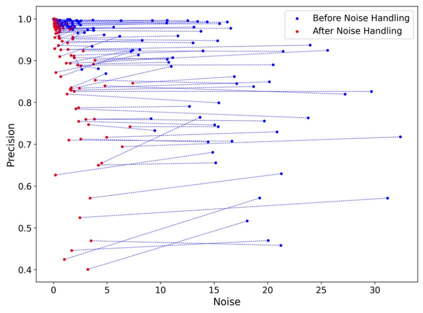
- Individual improvements per noise-handling technique and data-set, Section 8.5.1.1.
- Effect size of the improvements per technique, Section 8.5.1.2.
- Significance analysis of the effects per technique, Section 8.5.1.3.



(a) after using MF.



(b) after using CF.



(c) after using DB.

Figure 8.4: Precision scores after applying the treatment. The x-axis corresponds to the ratio of class noise, whereas the y-axis corresponds to the mean Precision scores. Blue marks represent the mean Precision scores before applying the treatment and the ratio of class noise. Red marks represent the mean performance score after applying the treatment and the new ratio of class noise.

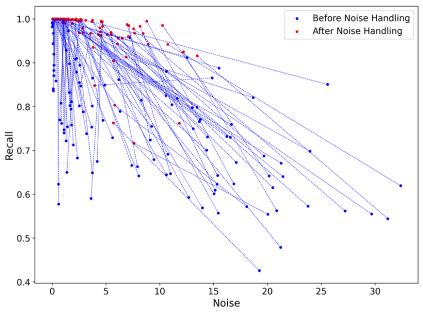
8.5.1.1 Individual improvements

Figure 8.4 illustrates the impact of each technique on the Precision measure across the examined study subjects. In general, we can observe that both MF and CF consistently improve the Precision measure of the RF model in the majority of the study subjects. This observation is supported by the diagonal upward trend observed in most of the lines depicted in Figure 8.4. Specifically, we can observe that:

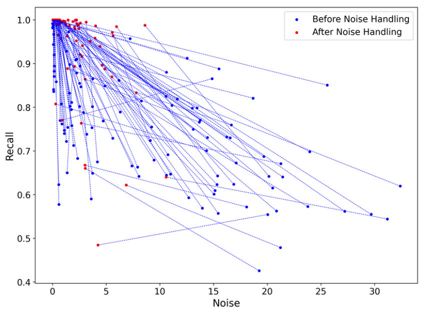
- MF resulted in a Precision improvement for 100/110 study subjects.
- CF resulted in a Precision improvement for 74/110 study subjects.
- DB resulted in a Precision improvement for 34/110 study subjects.

These observations imply that using the MF or CF techniques would consistently lead to an improvement in Precision.

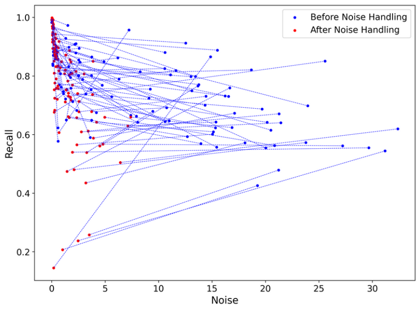
Another important observation is the effect of the techniques on the class noise



(a) after using MF.



(b) after using CF.



(c) after using DB.

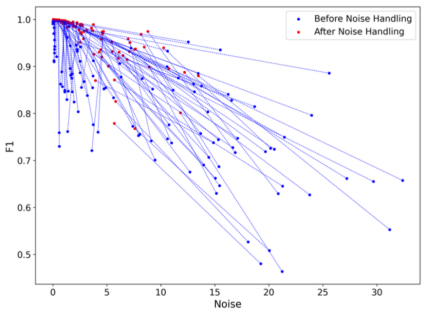
Figure 8.5: Recall scores after applying the treatment. The x-axis corresponds to the ratio of class noise, whereas the y-axis corresponds to the mean Precision scores. Blue marks represent the mean Precision scores before applying the treatment and the ratio of class noise. Red marks represent the mean performance score after applying the treatment and the new ratio of class noise.

ratio – i.e., how many contradictory entries they actually remove/correct. There, we can observe that the three techniques are consistent in reducing the ratio of class noise. Specifically, we found that:

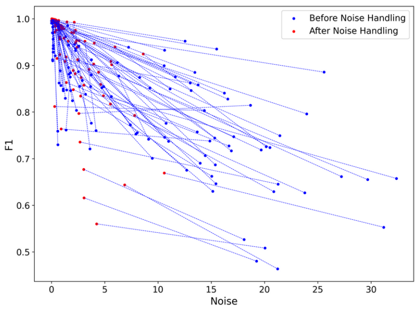
- MF reduced the ratio of class noise in 87/110 study subjects.
- CF reduced the ratio of class noise in 103/110.
- DB reduced the ratio of class noise in all study subjects.

Figure 8.5 illustrates the impact of each technique on the Recall measure across the examined study subjects. In general, we can observe that MF and CF are consistent in the effect and have a positive impact on the Recall measure in the majority of study subjects – most lines are diagonal in an upward direction. On the other hand, DB is less consistent in its effect on Recall. Specifically, we found that:

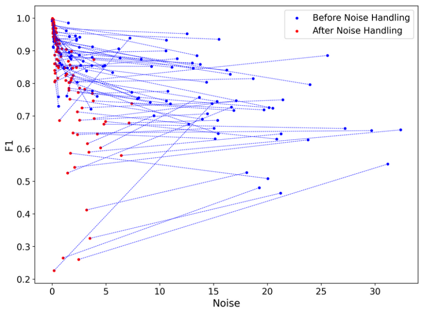
- MF resulted in a Recall improvement for 109/110 study subjects.
- CF resulted in a Recall improvement for 108/110 study subjects.



(a) after using MF.



(b) after using CF.



(c) after using DB.

Figure 8.6: F1 scores after applying the treatment. The x-axis corresponds to the ratio of class noise, whereas the y-axis corresponds to the mean Precision scores. Blue marks represent the mean Precision scores before applying the treatment and the ratio of class noise. Red marks represent the mean performance score after applying the treatment and the new ratio of class noise.

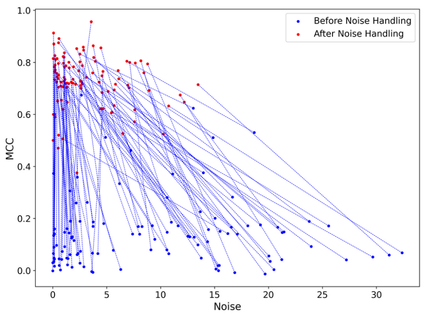
- DB resulted in a Recall improvement for 76/110 study subjects.

The above observations imply that MF and CF leads to a more consistent improvement in Recall compared to DB.

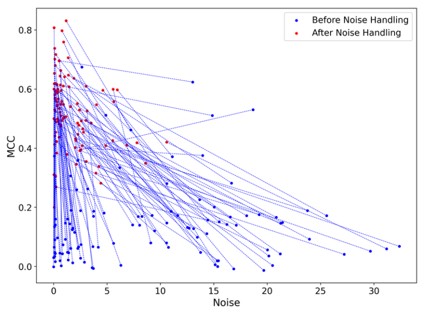
Figure 8.6 illustrates the impact of each technique on the F1 measure across the examined study subjects. Consistent with the findings observed for Precision and Recall, it can be observed that both MF and CF lead to a consistent improvement in F1. In contrast, applying DB does not consistently lead to an improvement in F1. Specifically, we found that:

- MF resulted in an F1 improvement for 110/110 study subjects.
- CF resulted in an F1 improvement for 109/110 study subjects.
- DB resulted in an F1 improvement for 76/110 study subjects.

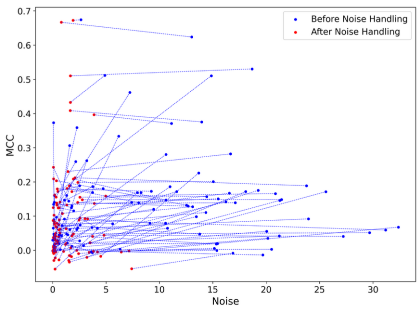
The above observations imply that MF and CF leads to a more consistent improvement in F1 compared to DB.



(a) after using MF.



(b) after using CF.



(c) after using DB.

Figure 8.7: MCC scores after applying the treatment. The x-axis corresponds to the ratio of class noise, whereas the y-axis corresponds to the mean Precision scores. Blue marks represent the mean Precision scores before applying the treatment and the ratio of class noise. Red marks represent the mean performance score after applying the treatment and the new ratio of class noise.

Figure 8.7 presents the impact of each technique on the MCC measure for each study subject. From the Figure, we can observe that all techniques are consistent in the effect on MCC, but in opposite directions. In general, we can observe that both MF and CF exhibit a consistent positive impact on MCC, while DB consistently leads to a negative improvement in MCC.

- MF resulted in an MCC improvement for 110/110 study subjects.
- CF resulted in an MCC improvement for 108/110 study subjects.
- DB resulted in an MCC improvement for 33/110 study subjects only.

These observations imply that applying MF or CF to the training data can consistently lead to an improvement in MCC.

8.5.1.2 Descriptive statistics

Figure 8.8 is a bar plot that visualizes the mean percentages of Precision, Recall, and F1 scores before and after applying the three treatment levels respectively.

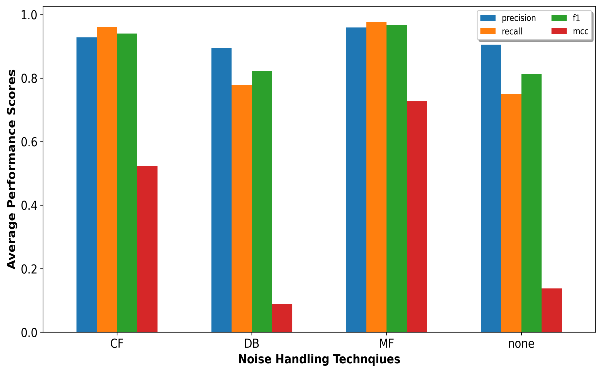


Figure 8.8: Mean Precision, Recall, F1, and MCC after using each noise-handling technique on build outcomes data

The x-axis represents the treatment levels, and the y-axis corresponds to the three dependent variable values.

Applying the three class noise-handling techniques on the control group data had the following impact on Precision, on average:

- MF resulted in improving Precision from 90% to 96%.
- CF resulted in improving Precision from 90% to 93%.
- DB resulted in a slight decrease in Precision from 90% to 89%.

Applying any of the three class noise-handling techniques to the control group data resulted in improved Recall values. Specifically, on average:

- MF resulted in improving Recall from 76% to 98%.
- CF resulted in improving Recall from 76% to 96%.
- DB resulted in improving Recall from 76% to 78%.

Applying any of the three class noise-handling techniques to the control group data resulted in improved F1 values. Specifically, on average:

- MF resulted in improving F1 from 82% to 97%.
- CF resulted in improving F1 from 82% to 94%.
- DB resulted in no improvement in F1.

Applying the three class noise-handling techniques on the control group data had the following impact on MCC, on average:

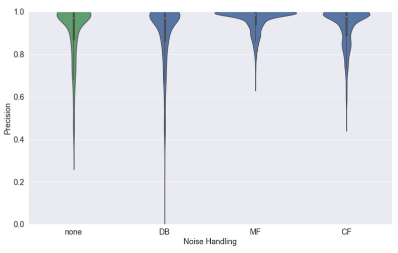
- MF resulted in improving MCC from 0.13 to 0.58.
- CF resulted in improving MCC from 0.13 to 0.52.

- DB resulted in a deterioration of MCC from 0.13 to 0.08.

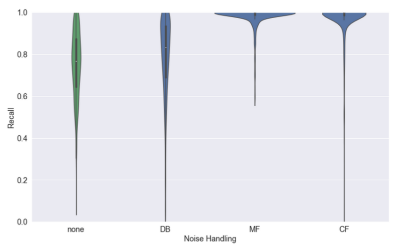
To gain a better understanding of the impact of each class noise handling technique, we plotted the distribution of each dependent variable for all the study subjects. The violin-plot graphs, namely Figures 8.9(a), 8.9(b), 8.9(c), and 8.9(d), illustrate the distribution of the dependent variables for the three treatment levels (MF, CF, and DB) as well as the control group data, labeled as ”none”. Four observations can be made from the four graphs:

- After applying MF and CF, the distribution of the Precision values become less dispersed and more concentrated around the median.
- After applying MF, the distribution of the Recall and F1 values became less dispersed. However, after applying DB and CF, the distribution of the F1 values became more dispersed, leading to a wider range of values.
- After applying the three treatment levels, the distribution of the MCC values became less dispersed.
- After applying MF and CF, the distribution of the MCC values became above 0.40 for the majority of cases.

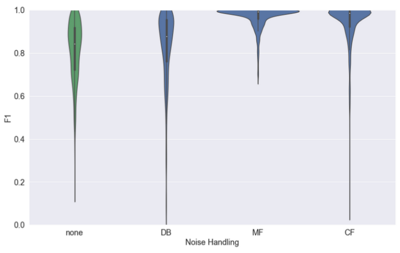
These observations were further supported by examining the descriptive statistics of the dependent variables summarized in Tables 8.16, 8.17, 8.18, 8.19.



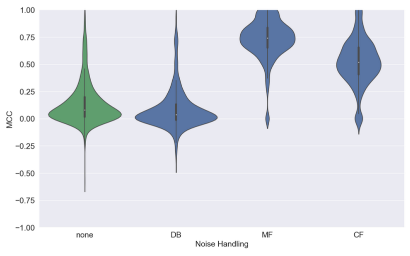
(a) Precision.



(b) Recall.



(c) F1.



(d) MCC.

Figure 8.9: Distribution of dependent variables for the three noise-handling techniques.

Table 8.7: The Shapiro-Wilk analysis results for each dependent variable after applying the noise-handling techniques to the build data.

Dependent variables	**Control group**	**DB**	**CF**	**MF**
Precision	Stats= 0.74, p<0.05	Stats= 0.73, p<0.05	Stats= 0.8, p<0.05	Stats= 0.79, p<0.05
Recall	Stats= 0.98, p<0.05	Stats= 0.89, p<0.05	Stats=5, p<0.05	Stats= 0.5, p<0.05
F1	Stats= 0.95, p<0.05	Stats=0.83 , p<0.05	Stats= 0.68, p<0.05	Stats= 0.69, p<0.05
MCC	Stats= 0.82, p<0.05	Stats= 0.75, p<0.05	Stats= 0.995, p=0.95	Stats= 0.96, p<0.05

In general, the descriptive statistics suggest that applying MF and CF leads to improved predictions of build outcomes. While the results suggest that DB improves the prediction performance for builds that will pass (improved Recall and F1), it tends to negatively affect the prediction accuracy for builds that will fail (lower MCC).

8.5.1.3 Hypotheses testing

To evaluate the hypotheses, we begin by checking the assumption of normality for the distribution of the four dependent variables. The Shapiro-Wilk test was carried out for testing the assumption of normality. As can be seen from Table 8.7, the null hypotheses of normality for the four dependent variables can be rejected (*p*-value <0.05). Since we have issues with normality in the four dependent variables, we decided to run a non-parametric test for comparing the difference between each dependent variable under the three treatment levels and the control group.

Table 8.8 summarizes the statistical comparison results between the four dependent variables for the three treatment levels and the control group using the Kruskal-Wallis analysis test. Each column provides the test statistics and the associated *p*-value for one dependent variable separately. The results in Table 8.8 reveal that there is a statistically significant difference between the four dependent variables (*p*<0.05).

Table 8.8: Statistical comparison results between the dependent variables after applying the treatment levels on the build data.

	Precision	**Recall**	**F1**	**MCC**
Kruskal Wallis H	12.74	241.89	166.47	334.17
Sig.	p<0.05	p<0.05	p<0.05	p<0.05

Table 8.9 presents the results of conducting the Mann-Whitney test to evaluate the twelve hypotheses in Experiment 1. The pairwise comparisons between the four dependent variables indicate a significant difference between the control group and the experimental subjects exposed to MF (Precision:

statistics = 4650, Recall: statistics = 420, F1: statistics = 974, MCC: statistics = 444, all with $p<0.05$). These findings provide statistical evidence to reject the null hypotheses H_{0p_m}, H_{0r_m}, H_{0f_m}, *and* H_{0m_m}, suggesting that MF has a significant impact on the predictive performance of the model for Build outcome prediction.

Table 8.9: Pairwise comparison between the dependent variables for each treatment level and the control group using the Mann-Whitny U test

	DB	**CF**	**MF**
Precision	Stats= 6147, p=0.49	Stats= 5619 p= 0.64	Stats= 4650 p<0.05
Recall	Stats= 4712 p<0.05	Stats= 829 p<0.05	Stats= 420 p<0.05
F1	Stats= 5020 p= 0.07	Stats= 1831 p<0.05	Stats= 974 p<0.05
MCC	Stats= 7698 p<0.05	Stats= 58 p<0.05	Stats= 444 p<0.05

Similarly, the results suggest that there is a statistically significant difference between the Recall, F1, and MCC scores achieved when training a model on the control group subjects and the experimental subjects exposed to CF (Recall: statistics= 829 with, F1: statistics= 1831, MCC: statistics= 58 with $p<0.05$). We also found a statistically significant difference between the Recall and MCC achieved when training on the control group subjects and the experimental subjects exposed to DB (Recall: statistics= 4712 with, MCC: statistics= 7698 with $p<0.05$). These results suggest that there is statistical evidence to support the rejection of the null hypotheses H_{0r_d}, H_{0r_c}, H_{0f_c}, H_{0m_d}, *and* H_{0m_c}. However, no statistical evidence was found to support the rejection of the hypotheses H_{0p_d}, H_{0p_c}, *and* H_{0f_d}.

To summarize, in the context of build outcome predictions, MF has a statistically significant positive impact on the four dependent variables. CF has a statistically significant positive impact on Recall, F1, and MCC. DB has a statistically significant positive impact on Recall and a statistically significant negative impact on MCC.

8.5.2 The impact of class noise-handling on predicting code change requests (RQ2)

To address the question "What is the impact of class noise handling on predicting code change requests?", we structure our results into two analyses:

- Effect size of improvements per technique, Section 8.5.2.1.
- Significance analysis of the effects per technique, Section 8.5.3.

8.5.2.1 Descriptive statistics

To evaluate the impact of the three noise handling techniques on the performance of a model for predicting negative review comments, we calculate the descriptive statistics of the four dependent variables after training on the control group data and each treatment level data respectively. Figure 8.10 is a bar plot that visualizes the mean percentages of Precision, Recall, F1, and MCC variables. The bar plot illustrates an improvement in the four variables. Specifically:

Applying the three class noise-handling techniques on the control group data had the following impact on Precision, on average:

- MF resulted in improving Precision from 34% to 83%.
- CF resulted in improving Precision from 34% to 70%.
- DB resulted in improving Precision from 34% to 39%.

Applying the three class noise-handling techniques on the control group data had the following impact on Recall, on average:

- MF resulted in improving Recall from 15% to 48%.
- CF resulted in improving Recall from 15% to 56%.
- DB resulted in improving Recall from 15% to 25%.

applying the three class noise-handling techniques on the control group data had the following impact on F1, on average:

- MF resulted in improving F1 from 18% to 54%.
- CF resulted in improving F1 from 18% to 60%.
- DB resulted in improving F1 from 18% to 27%.

applying the three class noise-handling techniques on the control group data had the following impact on MCC, on average:

- MF resulted in improving MCC from -0.03 to 0.57.
- CF resulted in improving MCC from -0.03 to 0.61.
- DB resulted in improving MCC from -0.03 to 0.17.

To better understand the impact of each class noise handling technique, we plotted the distribution of each dependent variable before and after applying the treatment levels to the control group data. Figures 8.11(a), 8.11(b), 8.11(c), and 8.11(d) show four violin-plot graphs that illustrate the distribution of each dependent variable for each treatment level. Three observations can be made from the violin-plot graphs:

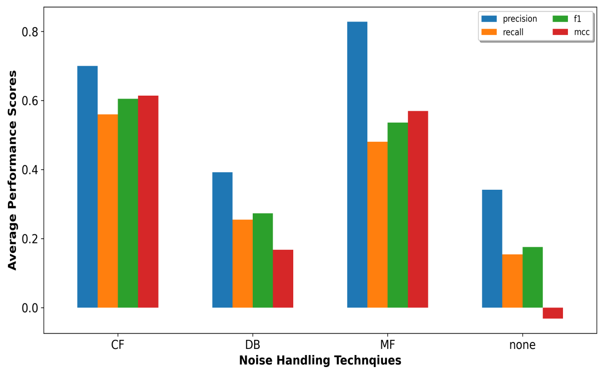
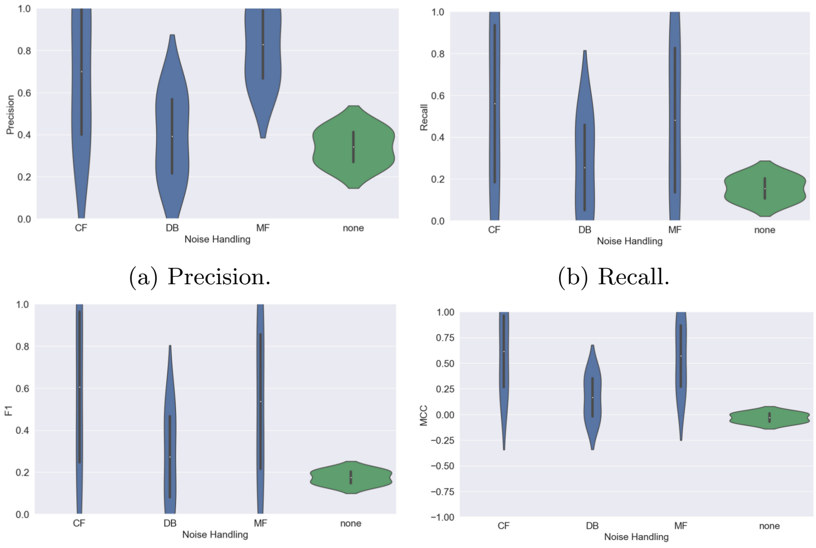


Figure 8.10: Mean Precision, Recall, F1, and MCC after using each noise-handling technique on code review data



(a) Precision.

(b) Recall.

(c) F1.

(d) MCC.

Figure 8.11: Distribution of dependent variables for the three noise-handling techniques.

- In terms of the Precision values, applying MF leads to the least dispersion compared to CF and DB.

- In terms of the Recall and F1 values, the distribution after applying the CF and MF ranges from 0 to 1 for some folds in the study subjects. Thus, there is a wide disparity in the distribution of these variables.

- In terms of the MCC values, applying DB leads to the least dispersion compared to MF and CF.

Table 8.10 shows the descriptive statistics of the dependent variables before and after applying the treatment. The table describes the mean and standard deviation (SD) of the dependent variables across the ten folds for every study subject. The data in the table shows that both MF and CF improved the four dependent variables in both study subjects, whereas DB improved the four variables in one study subject, and slightly reduced them in the other subject.

Table 8.10: Descriptive statistics of the dependent variables for an RF model before and after applying the treatment levels

			Precision		**Recall**		**F1**		**MCC**	
Noise algorithm	**project**	N	Mean	SD	Mean	SD	Mean	SD	Mean	SD
None	googlesources	10	0.41	0.41	0.2	0.26	0.2	0.22	0.01	0.28
	wireshark	10	0.27	0.1	0.11	0.05	0.15	0.06	-0.07	0.08
MF	googlesources	10	0.99	0.04	0.83	0.31	0.86	0.28	0.87	0.24
	wireshark	10	0.67	0.41	0.14	0.13	0.22	0.18	0.27	0.2
CF	googlesources	10	1.0	0.0	0.94	0.12	0.96	0.07	0.96	0.07
	wireshark	10	0.4	0.52	0.18	0.25	0.25	0.33	0.27	0.35
DB	googlesources	10	0.57	0.4	0.46	0.36	0.47	0.35	0.35	0.41
	wireshark	10	0.22	0.26	0.05	0.08	0.08	0.12	-0.02	0.15

In general, the descriptive statistics suggest that applying any of the three noise-handling techniques improves the performance of an ML model for build outcome predictions.

8.5.3 Hypotheses testing

To evaluate the hypotheses, we ran a Shapiro-Wilk analysis test to check the assumption of normality for the distribution of the dependent variables. Table 8.12 summarizes the normality test results for the four dependent variables.

Table 8.11: The Shapiro-Wilk analysis results for each dependent variable after applying the noise-handling techniques to the code review data.

Dependent variables	**Control group**	**DB**	**CF**	**MF**
Precision	Stats= 0.82 , p<0.05	Stats= 0.86, p<0.05	Stats= 0.58, p<0.05	Stats= 0.59, p<0.05
Recall	Stats= 0.72, p<0.05	Stats=0.76 , p<0.05	Stats= 0.79, p<0.05	Stats= 0.79, p<0.05
F1	Stats= 0.86, p<0.05	Stats= 0.8, p<0.05	Stats= 0.76, p<0.05	Stats= 0.82, p<0.05
MCC	Stats= 0.87, p<0.05	Stats=0.85 , p<0.05	Stats= 0.75, p<0.05	Stats= 0.86, p<0.05

Since we have issues in normality for the four dependent variables (p-value ¡0.05), we decided to compare the differences between each dependent variable under the three treatment levels and the control group with the aid of a non-parametric statistical analysis test.

Table 8.12 summarizes the statistical comparison results between the four dependent variables for the MF, CF, DB, and the control group using the Kruskal-Wallis test. The results from Table 8.12 reveal a statistically significant difference (p ¡ 0.05) between the four dependent variables. This suggests that the ML model's predictive performance for code change request predictions is different when trained on the control group compared to the experimental groups.

Table 8.12: Statistical comparison results between the dependent variables after applying the treatment levels on the code review data.

	Precision	**Recall**	**F1**	**MCC**
Kruskal Wallis H	Stats= 20.02	Stats= 11.06	Stats= 14.09	Stats= 34.89
Sig.	p<0.05	p<0.05	p<0.05	p<0.05

In addition, the results from Table 8.13 indicate a statistically significant difference (with p<0.05 for all pairwise comparisons) among the four dependent variables when training the ML model for build outcome predictions on data that has been exposed to MF and CF, as opposed to data that has not been exposed to any treatment.

Similarly, the results show that there is a statistically significant difference between the Recall values achieved when training the model on data that has been exposed to DB and those achieved when training on the control group data (Stats = 199,5, p<0.05). On the other hand, there was no statistically significant difference observed when comparing the Precision, F1, and MCC variables attained when training a model on data that has been exposed to DB and the control group data (p>0.05).

These results suggest that there is statistical evidence to support the rejection of the null hypotheses H_{0p_m}, H_{0p_c}, H_{0r_m}, H_{0r_c}, H_{0r_d}, H_{0f_m}, H_{f_c}, H_{0m_m}, *and* H_{0m_c}. In contrast, we did not find an evidence to support the rejection of the hypotheses H_{0p_d}, ${}_{0f\_d}$, *and* H_{0MCC_d}.

To summarize the results, MF and CF have a statistically significant impact on Precision, Recall, F1, and MCC. However, DB is not has a statistically significant effect on Recall only.

Table 8.13: Pairwise comparison between the dependent variables for each treatment level and the control group using the Mann-Whitney U test

	DB	**CF**	**MF**
Precision	Stats= 206.5, p=87	Stats= 124 p<0.05	Stats= 62, p<0.05
Recall	Stats= 199.5 p<0.05	Stats= 119 p<0.05	Stats= 109 p<0.05
F1	Stats= 194.5 p= 0.89	Stats= 113 p<0.05	Stats= 82.5 p<0.05
MCC	Stats= 151.5 p= 0.19	Stats= 45.0 p<0.05	Stats= 27.5 p<0.05

8.6 Discussion

In this section, we answer the two research questions and provide insights into the characteristics of lines of code that were identified as noisy by the three examined class noise-handling techniques.

8.6.1 RQ1- What is the impact of applying class noise-handling techniques on predicting the outcome of builds in continuous integration?

The evaluation results of this study reveal that applying removal-based techniques to the training data have a consistent positive impact on the predictive performance of the ML model for build outcome predictions. Specifically, the more conservative technique, known as the majority filter, was found to be more effective compared to the aggressive technique, referred to as the consensus filter. This suggests that by employing techniques that preserve a larger portion of the data, the model benefits from a more comprehensive analysis of the underlying patterns and trends, resulting in improved predictions. In addition, the evaluation showed that leaving the noise intact and relying on the tolerance capability of the model leads to a higher predictive performance of build outcomes compared to when applying the domain-knowledge based technique to the training data. This can be associated to several factors. Firstly, the domain-knowledge-based technique These findings align with previous studies that have advocated for the use of less rigid noise removal techniques to enhance the performance of ML models [176]. By employing techniques that preserve a larger portion of the data, the model benefits from a more comprehensive understanding of the underlying patterns and trends, resulting in improved predictions.

8.6.2 RQ2: What is the impact of applying class noise-handling techniques on predicting code change requests?

In line with the evaluation results for RQ1, our findings demonstrate that the removal-based techniques consistently improve the predictive performance of an ML model for change request predictions. Additionally, we observed that the domain-knowledge based technique improves the overall performance of the model, albeit to a lower extent compared to the removal-based techniques. Notably, the improvements achieved by applying the domain-knowledge technique were in the prediction of comments that request a code change, rather than comments that are accepted for integration. This was materialized in the significant improvements gained in MCC, but could not be found in F1.

However, it is important to interpret the evaluation results for both RQ1 and RQ2 in light of the noise ratios present in the data. In this study, the highest observed noise ratio among all analyzed projects did not exceed 40%. This implies that the findings may vary if the ratio of class noise exceeds this threshold. According to Teng [20], when applying noise-handling techniques to data with class noise exceeding 40%, the improvement in accuracy becomes inconsistent. Therefore, it is crucial to consider the noise ratio when applying these techniques in practice.

8.6.3 Characteristics of noisy lines of code

To gain a better understanding about the nature of lines of code that are classified as noisy, we observed patterns in the analyzed lines of code and sought to identify characteristics of lines that are treated as noisy by the three algorithms. Note that the analyzed lines of code are atomic parts of larger code-fragments. Hence, without the complete context, it is difficult to draw definitive conclusions about the characteristics of noisy lines. However, we discuss a few syntax-related characteristics observed among a sample of noisy lines identified by the three noise-handling techniques.

Table 8.14 provides a sample of 20 lines of code that we randomly selected and analyzed from the 'Wireshark' project. By examining the lines of code in the table, we can identify common patterns and trends about the characteristics of lines of code that are classified as noisy by the MF, CF, and dB.

One notable observation is that the lines labeled as noisy (True) by both the MF and CF, but not DB, tend to be lengthy and encompass logical conditions denoted by the presence of '&&'. These lines often involve logical operations, indicating a higher level of complexity in the logic of the program. The MF and CF algorithms seem to identify such lines as potentially noisy, possibly due to the increased likelihood of encountering change requests or inconsistencies in similar lines of code in the training data.

In contrast, the DB algorithm assigns the noisy label to lines that exhibit nested depth and contain short statements, primarily 'if' conditions. This suggests that the DB algorithm identifies noise within code blocks with intricate

Table 8.14: An excerpt of lines of code classified by the examined noise-handling techniques.

id	line of code	DB	CF	MF	Actual_class
1	\tFILE(READ "${CMAKE_CURRENT_BINARY_DIR} /version.h" VERSION_H_FILE_CONTENT)	False	True	True	0.0
2	cause_item = proto_tree_add_item(sub_tree, hf_gsm_r_chpc_cause, tvb, curr_offset, 1, ENC_NA)_	False	True	True	0.0
3	static void add_item(proto_tree *tree, int hf, tvbuff_t *tvb,	False	True	True	0.0
4	if ((octets_to_next_header == 0) && (version >= 0x0200) && (submessageId != SUBMESSAGE_PAD) && (submessageId != SUBMESSAGE_INFO_TS))	False	True	True	0.0
5	if (pinfo-fd-pkt_len = 60 frame_without_trailer 60) {	False	True	True	0.0
6	manuf = get_ether_name(tap_device-bd_addr)_	False	True	True	0.0
7	{ "unit", "ivi.unit",	True	False	False	0.0
8	\t\t\t\t\t\t\t"[unknown]",	True	False	False	1.0
9	if (req_resp) {	True	False	False	1.0
10	master_split_.show()_	True	False	False	1.0
11	return memcmp(buf, "SSTARRPC", 8) == 0_	True	False	False	1.0
12	offset += 1 + point_len_	True	False	False	0.0
13	}	True	True	True	1.0
14	{	True	True	True	1.0
15	p_add_proto_data(pinfo-pool, pinfo, proto_a_rr, 23, ppi)_	True	True	True	1.0
16	\t\t\tDISSECTOR_ASSERT(pdata != NULL)_	True	True	True	1.0
17	return NULL_	True	True	True	1.0
18	\t\t	False	False	False	0.0
19	case PAXOS_LEARN:	False	False	False	0.0
20	#include stdint.h	False	False	False	0.0

control flow structures. It considers the presence of nested conditions and short 'if' statements as indicators of potential noise, highlighting areas that might require closer attention or further investigation.

Interestingly, all three algorithms agree on certain characteristics that they classify as noisy. These include the presence of open and close brackets, function calls, and short return statements. Lines containing these elements are classified as noisy by all three algorithms, possibly due to the expectation of increased complexity in code fragments that follow open and closing brackets.

On the other hand, the algorithms unanimously classify lines that involve importing libraries and case statements as non-noisy. This agreement suggests that these elements are generally considered stable in their assigned class values, contributing to a lower likelihood of noise or error within these code segments.

By examining these common characteristics of lines classified by the MF, CF, and DB algorithms, we gain valuable insights into the characteristics of lines of code that are prone to be mislabeled.

8.6.4 Confounding factors

It is important to note that the effectiveness of the examined class noise handling techniques is subject to several factors related to the pre-processing activities performed on the training data. Three key activities were identified as potential influencers: data collection, feature extraction, and class balancing techniques.

Data collection as a crucial aspect of the study, relies on the reliability of the build records obtained from TravisTorrent. However, it is important to acknowledge that the accuracy of all the collected build job outcomes and commit hashes cannot be guaranteed. This introduces the possibility of inaccurate mappings between code changes and target class values within the data-set, potentially impacting the effectiveness of the noise handling techniques under examination.

Feature extraction. the choice of feature extraction technique can influence the predictive performance of the random forest (RF) model in both contexts. The specific method used to measure the frequency of tokens in the input code, such as word embeddings or TF-IDF, and variations in the configuration of the Bag-of-Words (BoW) model, including n-gram settings, may have an impact on the performance of the consensus filter (CF), majority filter (MF), and domain-knowledge based (DB) techniques. Exploring different feature extraction algorithms and BoW configuration parameters is an avenue for future research to better understand their influence on the effectiveness of the examined techniques.

Class balancing. the selection of a class balancing technique can also affect the performance of noise handling techniques. Different methods for balancing the classes should be empirically investigated to assess their impact on the effectiveness of CF, MF, and DB.

Considering these factors, it is crucial to interpret the study results with an awareness of the potential influence that these pre-processing activities. Thus, future research should focus on investigating different feature extraction techniques, and comparing different class balancing methods.

8.7 Research Validity

When analyzing the threats to validity of our study, we follow the framework recommended by Wohlin et al. [46] and discuss the validity in terms of external, internal, construct, and conclusion.

8.7.1 External validity threats

External validity refers to the degree to which the results can be generalized outside the context of the current study.

Sample size. the study subjects used in experiment 2 belong to two projects only. Hence, it is difficult to know whether the results drawn from this experiment can be generalized to the overall population of projects. However, we increase the likelihood of generalizability by using two different data sources to collect our sample projects and by randomly selecting a sample of code comments for annotation.

Programming Languages A potential threat to external validity in empirical SE research is the representativeness of the study subjects to other programming languages. However, we used a language-agnostic tool for extracting features from the study subjects. Hence, the probability that our findings apply to other programming languages in the two SE contexts increases.

8.7.2 Internal validity threats

Internal validity refers to the degree to which conclusions can be drawn about the causality between independent and dependent variables.

Instrumentation. A potential internal threat is the presence of undetected issues in the scripts we implemented and used for collecting code reviews and build data. This threat was controlled by carrying out a careful inspection of the scripts and testing them on small subsets.

Reuse of public data-sets. Since the analysis results of this study are based on public data-sets, we can not rule out the possibility of encountering erroneous data entries in the collected build and code review comment data-sets. However, we minimize this threat by manually validating a few of the collected review comments by our mining tools using the Gerrit web interface.

8.7.3 Construct validity threats

Construct validity refers to the degree to which experimental variables accurately measure the concepts they purport to measure.

Choice of the machine learning model. This study employed a random forest model as the final learner for evaluating the impact of class noise handling techniques in both SE contexts. It is difficult to assert whether tuning the parameters of the model or using different types of models (e.g., convolutional neural networks) would change the predictive performance results. To minimize this threat, we compared the performance of three different models in predicting build outcomes. The comparison results showed that random forest outperformed a two-dense layered neural network and an XGBoost model. Therefore, we decided to use random forest for measuring the effects of class noise-handling.

Data balancing technique. In this study, we used an over-sampling technique to deal with the problem of imbalanced training data. Using other techniques such as down-sampling or hybrid ones may change the reported results and, thereby, the conclusions. However, we chose to use an oversampling technique based on the recommendations of Mendoza et al. [177], which suggests that using oversampling yields better results than down-sampling and hybrid techniques.

8.7.4 Conclusion validity threats

Conclusion validity focuses on how sure we can be that the treatment we use really is related to the actual outcome we observe.

Class noise measurement. Our measurement of class noise is based on the ratio of contradictory entries in the data. However, inaccurate class values can also appear among entries that are not necessarily contradictory. This means that if we use different metrics for measuring class noise, we might reach different conclusions about the effect of each technique. However, our choice of using this metric for measuring class noise is motivated by our findings in previous research [173][98], where we identified a large number of contradictory entries in a regression testing data-set.

Compatibility of class noise handling techniques. This study examined the impact of three class noise handling techniques that handle class noise in different ways. That is, the removal-based techniques (MF and CF) utilize statistical measures to determine the accuracy of code labeling. In contrast, DB leverages domain knowledge regarding code changes to identify lines of code that require labeling. As a result, comparing the three techniques may pose challenges due to their inherent differences. Nonetheless, the objective of our study was to investigate how these diverse techniques impact the predictive performance of machine learning models when applied to software engineering data.

8.8 Conclusion and Future Work

In conclusion, our evaluation results highlight the consistent improvement in predictive performance achieved when applying the removal-based techniques for build outcome and code change request predictions. The domain-knowledge-based technique also shows potential in predicting code change requests, but not in build outcomes. In general, the effectiveness of the majority filter technique proved more effective than the consensus filter and domain-knowledge-based, particularly in build outcome predictions, suggesting that removing entries with class noise to a certain threshold leads to improved predictions in build outcome. On the other hand, the consensus filter technique proved slightly more effective than the majority filter in the context of code change request predictions. In addition, the results reveal that leaving the noise intact and relying on the model's tolerance capability outperformed a domain-knowledge-based technique for build outcome predictions.

There are several avenues for future work that would enhance our understanding of noise-handling techniques and their impact in CI contexts. Firstly, a larger sample of projects within the context of code change request predictions is needed to provide a more comprehensive understanding of the generalizability of the effectiveness of the techniques on predicting code change request prediction. Secondly, examining the impact of alternative noise-handling techniques beyond those examined in this study is needed. While this study focused on the Majority Filter, Consensus Filter, and Domain-knowledge-based techniques, there may be other techniques that can provide improved predictive performance for CI tasks. Thirdly, examining the effects of the examined noise-handling techniques under a higher ratio of class noise is needed. In this study, the highest ratio of class noise among all the analyzed projects did not exceed 40%. However, different projects or contexts may exhibit higher ratios of class noise. Thus, understanding how the examined techniques perform under higher levels of class noise would provide insights into their robustness and scalability. Finally, it would be insightful to examine whether the size of projects plays a role in influencing the accuracy improvement achieved by each noise-handling technique. As larger projects often involve a larger complexity of code and a higher number of potential noise sources, there is a probability that the examined techniques would behave differently when applied to projects of varying sizes. Thus, investigating whether the effectiveness of the techniques would vary based on project size is needed to understand the generalizability of the three examined techniques.

8.9 Appendix A

Table 8.15: The Evaluation Results for Comparing the Effectiveness of Random Forest, Extended Gradient Boosting, and Neural Network

index	**noise_alg**	**classifier**	**Precision**	**Recall**	**F1**	**MCC**
0	CF	RF	0.85	0.86	0.85	0.23
1	CF	nn	0.85	0.86	0.85	0.22
2	CF	xgb	0.83	0.91	0.84	0.17
3	DB	RF	0.83	0.83	0.8	0.12
4	DB	nn	0.83	0.64	0.69	0.15
5	DB	xgb	0.85	0.64	0.68	0.15
6	MF	RF	0.85	0.88	0.86	0.26
7	MF	nn	0.85	0.88	0.86	0.22
8	MF	xgb	0.83	0.9	0.84	0.15
9	none	RF	0.86	0.9	0.87	0.23
10	none	nn	0.86	0.88	0.87	0.2
11	none	xgb	0.85	0.92	0.87	0.17

Table 8.16: Descriptive statistics of the dependent variables for an RF model before applying any treatment levels.

Project	**Precision**		**Recall**		**F1**		**MCC**	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD
AcDisplay	0.47	0.02	0.55	0.03	0.51	0.02	0.06	0.04
DDT	0.83	0.02	0.55	0.15	0.66	0.11	0.05	0.08
HearthSim	1.0	0.0	0.62	0.14	0.76	0.1	0.04	0.03
HikariCP	0.91	0.03	0.66	0.2	0.75	0.15	0.17	0.17
Hydra	0.94	0.01	0.7	0.1	0.8	0.07	0.09	0.08
Hystrix	0.63	0.07	0.67	0.13	0.65	0.09	0.15	0.17
Jest	0.89	0.01	0.62	0.09	0.72	0.06	0.0	0.06
LittleProxy	0.9	0.01	0.69	0.11	0.78	0.08	0.06	0.05
MozStumbler	1.0	0.0	0.97	0.02	0.98	0.01	0.09	0.13
OpenRefine	0.96	0.03	0.86	0.06	0.91	0.04	0.33	0.22
ProjectRed	0.88	0.08	0.85	0.17	0.86	0.12	0.67	0.23
RoaringBitmap	0.99	0.0	0.74	0.08	0.85	0.05	0.15	0.05
Singularity	0.79	0.09	0.59	0.06	0.68	0.06	0.13	0.19
Dspace	0.97	0.0	0.99	0.01	0.98	0.01	0.08	0.16
auto	0.99	0.0	1.00	0.00	0.99	0.00	0.16	0.17
airlift	0.68	0.24	0.87	0.05	0.74	0.15	0.51	0.32
analytics-android	0.95	0.02	0.81	0.09	0.87	0.06	0.17	0.17
android	0.94	0.07	0.8	0.06	0.86	0.05	0.62	0.19
android-maven-plugin	0.96	0.01	0.82	0.16	0.88	0.11	0.17	0.16
assertj-android	0.92	0.05	0.57	0.18	0.69	0.14	0.38	0.17
basex	0.97	0.0	0.79	0.04	0.87	0.02	0.01	0.01
blueflood	0.93	0.03	0.67	0.07	0.77	0.05	0.14	0.13
blueprints	0.71	0.1	0.73	0.11	0.71	0.04	0.16	0.19
bnd	0.93	0.02	0.64	0.08	0.76	0.05	0.14	0.07
brightspot-cms	0.87	0.08	0.85	0.06	0.85	0.04	0.51	0.2
cas-addons	1.0	0.0	0.91	0.02	0.95	0.01	0.13	0.13
cassandra-reaper	0.88	0.04	0.68	0.12	0.76	0.1	0.07	0.14
ccw	0.76	0.02	0.72	0.05	0.74	0.03	0.08	0.08

checkstyle	1.0	0.0	0.87	0.06	0.93	0.03	0.04	0.05
cloudify	0.82	0.02	0.56	0.11	0.66	0.07	0.04	0.07
core	1.0	0.0	0.58	0.05	0.73	0.04	0.02	0.03
dagger	0.97	0.01	0.77	0.05	0.86	0.03	0.05	0.07
dropwizard	0.99	0.01	0.75	0.06	0.85	0.04	0.09	0.08
dynjs	0.98	0.01	0.73	0.16	0.83	0.1	0.17	0.27
error-prone	1.0	0.0	0.99	0.01	0.99	0.0	0.07	0.05
frontend-maven-plugin	0.98	0.01	0.8	0.06	0.88	0.04	0.19	0.12
go-lang-idea-plugin	0.97	0.01	0.76	0.14	0.85	0.09	0.13	0.08
goclipse	0.99	0.0	0.8	0.06	0.89	0.04	0.03	0.02
gpslogger	0.96	0.01	0.77	0.08	0.85	0.05	0.18	0.11
hivemall	0.99	0.0	0.88	0.06	0.93	0.03	0.06	0.06
htm.java	1.0	0.0	0.88	0.09	0.93	0.05	0.28	0.3
idea-gitignore	0.85	0.02	0.64	0.11	0.73	0.09	0.04	0.08
jInstagram	1.0	0.0	0.76	0.14	0.85	0.1	0.17	0.09
jPOS	1.0	0.0	0.79	0.1	0.88	0.06	0.0	0.04
jade4j	0.98	0.01	0.65	0.23	0.76	0.18	0.02	0.07
javaslang	1.0	0.0	0.91	0.13	0.95	0.08	0.26	0.22
jcabi-aspects	0.89	0.04	0.65	0.17	0.74	0.12	0.19	0.14
jcabi-github	0.76	0.14	0.77	0.08	0.76	0.08	0.23	0.34
jcabi-http	0.86	0.03	0.62	0.12	0.72	0.09	-0.01	0.1
jedis	0.99	0.0	0.85	0.13	0.91	0.08	0.19	0.12
jmeter-plugins	0.92	0.06	0.96	0.02	0.94	0.04	0.46	0.43
jmonkeyengine	0.99	0.0	0.8	0.02	0.89	0.01	0.1	0.06
jmxtrans	0.98	0.0	0.76	0.07	0.86	0.04	0.07	0.07
joda-time	1.0	0.0	0.88	0.05	0.94	0.03	0.02	0.04
jodd	0.99	0.0	0.7	0.22	0.8	0.16	0.11	0.13
jphp	0.9	0.04	0.64	0.14	0.75	0.09	0.15	0.22
jsonld-java	1.0	0.0	0.94	0.04	0.97	0.02	0.13	0.09
jsprit	1.0	0.0	0.89	0.06	0.94	0.03	0.16	0.09
keywhiz	1.0	0.0	0.99	0.01	1.0	0.0	-0.0	0.0
lenskit	0.99	0.0	0.81	0.12	0.88	0.09	0.01	0.04
less4j	0.92	0.03	0.85	0.03	0.89	0.02	0.17	0.2
logback	0.98	0.01	0.73	0.08	0.83	0.06	0.08	0.06
lorsource	0.98	0.0	0.72	0.09	0.83	0.06	0.05	0.05
maven-git-commit-id-plugin	0.92	0.02	0.64	0.13	0.75	0.1	0.15	0.14
metrics	0.91	0.05	0.8	0.08	0.85	0.04	0.37	0.18
mockito	0.97	0.01	0.86	0.11	0.91	0.06	0.19	0.1
mybatis-3	1.0	0.0	0.75	0.1	0.85	0.07	0.07	0.04
nokogiri	0.71	0.11	0.76	0.12	0.73	0.09	0.28	0.24
nutz	0.66	0.03	0.61	0.06	0.63	0.04	0.01	0.08
okhttp	0.76	0.03	0.69	0.05	0.72	0.03	-0.01	0.09
onebusaway-android	1.0	0.0	0.93	0.06	0.96	0.03	0.16	0.16
openwayback	1.0	0.0	0.97	0.02	0.99	0.01	0.14	0.13
owner	0.99	0.0	0.81	0.06	0.89	0.04	0.04	0.08
p6spy	0.84	0.16	0.82	0.08	0.81	0.06	0.53	0.28
parceler	0.99	0.0	0.68	0.1	0.8	0.08	0.07	0.05
pdfsam	0.8	0.14	0.56	0.2	0.65	0.19	0.02	0.22
play-authenticate	0.97	0.01	0.91	0.08	0.94	0.05	0.36	0.22
psi-probe	1.0	0.0	0.92	0.08	0.96	0.04	0.14	0.21
pushy	1.0	0.0	0.99	0.01	0.99	0.0	0.37	0.33
querydsl	0.98	0.01	0.75	0.12	0.85	0.08	0.05	0.08
quickml	0.98	0.01	0.71	0.06	0.82	0.04	0.12	0.1
qulice	0.98	0.0	0.8	0.05	0.88	0.03	-0.01	0.05
restlet-framework-java	0.57	0.06	0.54	0.16	0.55	0.11	0.06	0.15
retrofit	0.99	0.0	0.89	0.04	0.94	0.02	0.15	0.1

rewrite	0.76	0.06	0.57	0.29	0.63	0.2	0.19	0.27
rexster	0.99	0.01	0.79	0.15	0.88	0.09	0.12	0.17
robospice	0.46	0.06	0.48	0.11	0.46	0.07	0.04	0.11
rultor	0.72	0.02	0.62	0.12	0.66	0.07	0.07	0.04
rxjava-jdbc	1.0	0.0	0.84	0.04	0.91	0.02	0.03	0.04
selendroid	0.73	0.03	0.68	0.12	0.7	0.08	0.12	0.09
seyren	0.99	0.01	0.73	0.07	0.84	0.05	0.14	0.09
sms-backup-plus	0.99	0.0	0.77	0.09	0.86	0.06	0.0	0.03
spark	0.99	0.0	0.83	0.04	0.9	0.02	0.08	0.07
spring-cloud-config	0.95	0.02	0.62	0.14	0.74	0.1	0.02	0.08
springside4	0.73	0.07	0.56	0.15	0.63	0.12	0.17	0.15
storio	0.97	0.02	0.65	0.07	0.78	0.05	0.17	0.11
storm	0.52	0.13	0.57	0.17	0.53	0.1	0.17	0.19
structr	0.57	0.12	0.43	0.11	0.48	0.1	0.18	0.17
stubby4j	0.75	0.07	0.6	0.09	0.66	0.06	0.2	0.12
thredds	0.74	0.03	0.64	0.08	0.69	0.06	-0.0	0.11
traccar	1.0	0.0	0.84	0.02	0.91	0.01	0.05	0.03
truth	0.97	0.02	0.74	0.06	0.84	0.04	0.26	0.14
twilio-java	0.85	0.03	0.67	0.08	0.75	0.05	0.14	0.11
u2020	0.98	0.01	0.75	0.1	0.85	0.07	0.13	0.13
unirest-java	0.93	0.01	0.59	0.07	0.72	0.05	-0.0	0.03
waffle	1.0	0.0	0.86	0.06	0.92	0.03	0.1	0.08
webcam-capture	1.0	0.0	0.89	0.04	0.94	0.02	0.07	0.04
wire	1.0	0.0	0.98	0.02	0.99	0.01	0.03	0.05
xtreemfs	0.98	0.02	0.83	0.12	0.9	0.08	0.31	0.27
yobi	1.0	0.0	0.9	0.05	0.94	0.03	0.06	0.03

Table 8.17: Descriptive statistics of the dependent variables for an RF model after applying DB.

Project	**Precision**		**Recall**		**F1**		**MCC**	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD
AcDisplay	0.45	0.02	0.86	0.04	0.59	0.02	0.01	0.08
DDT	0.83	0.03	0.54	0.12	0.65	0.09	0.05	0.09
DSpace	0.98	0.0	0.9	0.09	0.93	0.05	0.05	0.05
HearthSim	0.99	0.0	0.92	0.06	0.96	0.04	0.0	0.02
HikariCP	0.87	0.01	0.85	0.07	0.86	0.04	-0.05	0.08
Hydra	0.93	0.01	0.92	0.02	0.92	0.01	0.06	0.07
Hystrix	0.57	0.03	0.61	0.07	0.59	0.05	-0.01	0.09
Jest	0.89	0.01	0.82	0.08	0.85	0.05	0.0	0.05
LittleProxy	0.89	0.01	0.68	0.06	0.77	0.04	0.04	0.06
MozStumbler	1.0	0.0	0.98	0.01	0.99	0.01	0.08	0.17
OpenRefine	0.91	0.01	0.83	0.06	0.87	0.04	-0.03	0.08
ProjectRed	0.91	0.1	0.81	0.16	0.85	0.11	0.67	0.22
RoaringBitmap	0.98	0.0	0.99	0.01	0.99	0.0	0.01	0.04
Singularity	0.79	0.06	0.66	0.08	0.71	0.05	0.14	0.14
DSpace	0.98	0.0	0.90	0.09	0.93	0.05	0.05	0.05
auto	0.99	0.0	0.88	0.03	0.93	0.02	0.06	0.06
airlift	0.63	0.23	0.14	0.12	0.23	0.17	0.16	0.18
analytics-android	0.94	0.01	0.87	0.08	0.9	0.05	0.13	0.15
android-maven-plugin	0.95	0.01	0.77	0.06	0.85	0.04	0.06	0.05
android	0.95	0.06	0.83	0.05	0.88	0.04	0.67	0.15

assertj-android	0.91	0.04	0.66	0.19	0.75	0.13	0.41	0.16
auto	0.99	0.0	0.88	0.03	0.93	0.02	0.06	0.06
basex	0.97	0.0	0.86	0.05	0.91	0.03	0.04	0.05
blueflood	0.89	0.01	0.81	0.06	0.85	0.03	0.02	0.06
blueprints	0.71	0.11	0.68	0.08	0.69	0.06	0.16	0.22
bnd	0.93	0.02	0.75	0.07	0.83	0.04	0.18	0.1
brightspot-cms	0.84	0.1	0.89	0.04	0.86	0.05	0.43	0.31
cas-addons	1.0	0.0	0.94	0.02	0.97	0.01	0.2	0.11
cassandra-reaper	0.89	0.03	0.71	0.13	0.79	0.1	0.09	0.13
ccw	0.76	0.03	0.81	0.07	0.78	0.04	0.09	0.13
checkstyle	1.0	0.0	0.99	0.01	1.0	0.0	-0.0	0.0
cloudify	0.84	0.02	0.57	0.07	0.68	0.05	0.04	0.05
core	1.0	0.0	0.68	0.08	0.81	0.05	0.02	0.04
dagger	0.97	0.01	0.83	0.05	0.9	0.03	0.07	0.09
dropwizard	0.98	0.0	0.72	0.05	0.83	0.04	-0.01	0.02
dynjs	0.98	0.01	0.85	0.09	0.91	0.05	0.11	0.12
error-prone	1.0	0.0	0.99	0.0	1.0	0.0	0.09	0.08
frontend-maven-plugin	0.96	0.01	0.92	0.06	0.94	0.03	0.03	0.12
go-lang-idea-plugin	0.96	0.02	0.72	0.14	0.82	0.1	0.03	0.18
goclipse	0.99	0.0	0.89	0.03	0.94	0.01	0.03	0.02
gpslogger	0.96	0.01	0.85	0.05	0.9	0.03	0.17	0.14
hivemall	0.99	0.0	0.89	0.06	0.94	0.04	-0.0	0.04
htm.java	0.99	0.0	0.96	0.04	0.98	0.02	0.04	0.06
idea-gitignore	0.83	0.02	0.74	0.19	0.77	0.13	-0.01	0.09
jInstagram	1.0	0.0	0.76	0.15	0.85	0.1	0.18	0.09
jPOS	1.0	0.0	0.88	0.09	0.93	0.05	0.0	0.03
jade4j	0.98	0.01	0.73	0.26	0.81	0.19	-0.02	0.08
javaslang	1.0	0.0	0.93	0.09	0.96	0.05	0.16	0.16
jcabi-aspects	0.83	0.03	0.74	0.13	0.78	0.08	-0.02	0.13
jcabi-github	0.65	0.05	0.56	0.1	0.6	0.08	-0.02	0.12
jcabi-http	0.85	0.04	0.66	0.11	0.74	0.08	-0.05	0.13
jedis	0.98	0.0	0.9	0.11	0.94	0.06	0.17	0.12
jmeter-plugins	0.86	0.09	0.61	0.29	0.69	0.22	0.14	0.4
jmonkeyengine	0.99	0.0	0.97	0.01	0.98	0.01	-0.0	0.02
jmxtrans	0.98	0.0	0.97	0.02	0.97	0.01	-0.0	0.03
joda-time	1.0	0.0	0.94	0.03	0.97	0.01	-0.01	0.01
jodd	0.99	0.0	0.78	0.18	0.86	0.11	0.07	0.07
jphp	0.91	0.03	0.77	0.16	0.82	0.11	0.21	0.24
jsonld-java	1.0	0.0	0.97	0.03	0.98	0.02	0.12	0.11
jsprit	1.0	0.0	0.92	0.06	0.95	0.03	0.01	0.04
keywhiz	1.0	0.0	1.0	0.0	1.0	0.0	-0.0	0.0
lenskit	0.99	0.0	0.87	0.08	0.92	0.05	0.02	0.04
less4j	0.89	0.01	0.73	0.06	0.8	0.04	-0.03	0.07
logback	0.98	0.0	0.83	0.06	0.9	0.04	0.08	0.06
lorsource	0.98	0.0	0.79	0.07	0.87	0.04	0.03	0.04
maven-git-commit-id-plugin	0.92	0.03	0.61	0.16	0.72	0.12	0.15	0.16
metrics	0.9	0.05	0.85	0.07	0.87	0.05	0.4	0.2
mockito	0.97	0.01	0.9	0.08	0.93	0.04	0.13	0.14
mybatis-3	0.99	0.0	0.68	0.04	0.8	0.03	0.01	0.06
nokogiri	0.71	0.16	0.47	0.32	0.53	0.26	0.23	0.29
nutz	0.65	0.02	0.64	0.06	0.64	0.04	-0.0	0.05
okhttp	0.76	0.02	0.74	0.06	0.75	0.03	-0.0	0.08
onebusaway-android	1.0	0.0	0.97	0.03	0.98	0.02	0.08	0.09
openwayback	1.0	0.0	0.99	0.01	1.0	0.0	0.24	0.27
owner	0.99	0.0	0.97	0.04	0.98	0.02	-0.01	0.02
p6spy	0.83	0.16	0.8	0.08	0.8	0.07	0.51	0.28

parceler	0.99	0.0	0.71	0.09	0.82	0.07	0.06	0.05
pdfsam	0.82	0.04	0.82	0.18	0.81	0.1	0.02	0.23
play-authenticate	0.95	0.02	0.88	0.11	0.91	0.06	0.12	0.17
psi-probe	1.0	0.0	0.96	0.03	0.98	0.02	0.21	0.19
pushy	1.0	0.0	0.99	0.01	0.99	0.0	0.1	0.24
querydsl	0.97	0.01	0.75	0.17	0.84	0.12	-0.02	0.05
quickml	0.98	0.01	0.76	0.05	0.85	0.03	0.12	0.12
qulice	0.98	0.0	0.78	0.05	0.87	0.03	-0.03	0.04
restlet-framework-java	0.52	0.31	0.24	0.39	0.26	0.37	0.09	0.45
retrofit	0.99	0.0	0.93	0.04	0.96	0.02	0.13	0.09
rewrite	0.78	0.08	0.48	0.35	0.54	0.28	0.21	0.3
rexster	0.99	0.01	0.86	0.1	0.92	0.06	0.14	0.17
robospice	0.47	0.14	0.26	0.07	0.32	0.07	0.02	0.14
rultor	0.69	0.06	0.5	0.08	0.58	0.06	-0.0	0.14
rxjava-jdbc	1.0	0.0	0.87	0.04	0.93	0.02	0.04	0.04
selendroid	0.75	0.07	0.54	0.12	0.62	0.09	0.09	0.14
seyren	0.98	0.0	0.81	0.05	0.89	0.03	0.02	0.04
sms-backup-plus	0.99	0.0	0.95	0.03	0.97	0.01	0.01	0.05
spark	0.99	0.0	0.82	0.08	0.89	0.05	0.05	0.05
spring-cloud-config	0.95	0.01	0.72	0.09	0.82	0.06	0.04	0.09
springside4	0.72	0.07	0.66	0.14	0.68	0.1	0.16	0.19
storio	0.98	0.02	0.66	0.08	0.79	0.05	0.19	0.11
storm	0.4	0.08	0.43	0.14	0.41	0.1	-0.02	0.15
structr	0.42	0.12	0.21	0.14	0.26	0.12	-0.0	0.1
stubby4j	0.75	0.1	0.57	0.09	0.64	0.08	0.2	0.19
thredds	0.74	0.04	0.63	0.09	0.68	0.06	-0.0	0.12
traccar	1.0	0.0	0.89	0.03	0.94	0.02	0.04	0.03
truth	0.93	0.0	0.98	0.02	0.95	0.01	-0.03	0.02
twilio-java	0.85	0.05	0.59	0.1	0.69	0.08	0.14	0.16
u2020	0.98	0.01	0.9	0.08	0.93	0.05	0.09	0.11
unirest-java	0.94	0.01	0.88	0.08	0.9	0.05	0.03	0.14
waffle	0.99	0.0	0.87	0.05	0.93	0.03	0.05	0.03
webcam-capture	0.99	0.0	0.89	0.08	0.94	0.05	0.07	0.05
wire	1.0	0.0	0.99	0.0	1.0	0.0	0.04	0.09
xtreemfs	0.96	0.0	0.97	0.02	0.96	0.01	0.0	0.05
yobi	1.0	0.0	0.91	0.05	0.95	0.03	0.03	0.05

Table 8.18: Descriptive statistics of the dependent variables for an RF model after applying MF.

Project	**Precision**		**Recall**		**F1**		**MCC**	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD
AcDisplay	0.84	0.05	0.72	0.08	0.77	0.04	0.62	0.06
DDT	0.9	0.02	0.96	0.05	0.93	0.02	0.61	0.04
DSpace	0.99	0.0	1.0	0.0	1.0	0.0	0.8	0.06
HearthSim	1.0	0.0	1.0	0.0	1.0	0.0	0.52	0.19
HikariCP	0.94	0.01	0.97	0.07	0.95	0.03	0.61	0.12
Hydra	0.97	0.01	1.0	0.0	0.98	0.0	0.78	0.05
Hystrix	0.85	0.04	0.9	0.07	0.87	0.03	0.69	0.06
Jest	0.95	0.01	0.99	0.01	0.97	0.01	0.68	0.08
LittleProxy	0.95	0.01	1.0	0.01	0.97	0.01	0.75	0.07
MozStumbler	1.0	0.0	1.0	0.0	1.0	0.0	0.78	0.31

OpenRefine	0.98	0.01	1.0	0.0	0.99	0.01	0.86	0.09
ProjectRed	1.0	0.01	0.97	0.03	0.98	0.02	0.96	0.04
RoaringBitmap	0.99	0.0	0.99	0.02	0.99	0.01	0.62	0.15
Singularity	0.9	0.04	0.96	0.04	0.93	0.02	0.72	0.11
DSpace	0.99	0.0	1.0	0.0	1.0	0.0	0.80	0.06
auto	1.00	0.0	1.0	0.0	1.0	0.0	0.78	0.14
airlift	0.9	0.06	0.85	0.16	0.87	0.11	0.8	0.16
analytics-android	0.98	0.01	1.0	0.0	0.99	0.01	0.85	0.09
android	0.94	0.02	0.97	0.03	0.95	0.02	0.83	0.05
android-maven-plugin	0.98	0.01	1.0	0.01	0.99	0.01	0.73	0.12
assertj-android	0.88	0.03	0.96	0.01	0.92	0.02	0.62	0.11
auto	1.0	0.0	1.0	0.0	1.0	0.0	0.78	0.14
basex	0.99	0.0	1.0	0.01	0.99	0.0	0.72	0.09
blueflood	0.95	0.01	1.0	0.0	0.97	0.0	0.71	0.05
blueprints	0.9	0.05	0.94	0.04	0.92	0.02	0.77	0.07
bnd	0.96	0.01	1.0	0.0	0.98	0.0	0.77	0.03
brightspot-cms	0.94	0.04	0.98	0.02	0.96	0.01	0.86	0.06
cas-addons	1.0	0.0	1.0	0.0	1.0	0.0	0.89	0.14
cassandra-reaper	0.95	0.02	0.99	0.0	0.97	0.01	0.77	0.1
ccw	0.92	0.02	0.98	0.01	0.95	0.01	0.8	0.05
checkstyle	1.0	0.0	1.0	0.0	1.0	0.0	0.81	0.12
cloudify	0.9	0.01	0.98	0.01	0.94	0.01	0.57	0.06
core	1.0	0.0	1.0	0.0	1.0	0.0	0.47	0.2
dagger	0.99	0.01	1.0	0.0	0.99	0.0	0.8	0.15
dropwizard	0.99	0.0	1.0	0.0	0.99	0.0	0.7	0.05
dynjs	0.99	0.01	1.0	0.0	0.99	0.0	0.79	0.13
error-prone	1.0	0.0	1.0	0.0	1.0	0.0	0.81	0.15
frontend-maven-plugin	0.98	0.0	1.0	0.0	0.99	0.0	0.7	0.06
go-lang-idea-plugin	0.99	0.01	1.0	0.0	0.99	0.0	0.8	0.12
goclipse	1.0	0.0	1.0	0.0	1.0	0.0	0.66	0.13
gpslogger	0.98	0.01	1.0	0.0	0.99	0.0	0.79	0.08
hivemall	1.0	0.0	1.0	0.0	1.0	0.0	0.8	0.15
htm.java	1.0	0.0	1.0	0.0	1.0	0.0	0.81	0.21
idea-gitignore	0.9	0.03	0.99	0.02	0.94	0.01	0.52	0.18
jInstagram	0.99	0.0	1.0	0.0	1.0	0.0	0.72	0.1
jPOS	1.0	0.0	1.0	0.0	1.0	0.0	0.76	0.2
jade4j	0.99	0.0	1.0	0.0	0.99	0.0	0.38	0.3
javaslang	1.0	0.0	1.0	0.01	1.0	0.01	0.79	0.2
jcabi-aspects	0.93	0.03	0.99	0.01	0.96	0.01	0.71	0.1
jcabi-github	0.91	0.07	0.97	0.02	0.94	0.05	0.8	0.15
jcabi-http	0.96	0.03	0.99	0.0	0.97	0.01	0.79	0.11
jedis	0.99	0.0	0.99	0.03	0.99	0.01	0.72	0.16
jmeter-plugins	0.98	0.02	0.97	0.05	0.97	0.03	0.85	0.14
jmonkeyengine	1.0	0.0	1.0	0.0	1.0	0.0	0.75	0.13
jmxtrans	0.99	0.0	1.0	0.0	1.0	0.0	0.82	0.1
joda-time	1.0	0.0	1.0	0.0	1.0	0.0	0.73	0.22
jodd	1.0	0.0	1.0	0.0	1.0	0.0	0.74	0.13
jphp	0.95	0.02	1.0	0.0	0.97	0.01	0.73	0.1
jsonld-java	1.0	0.0	1.0	0.0	1.0	0.0	0.76	0.1
jsprit	1.0	0.0	1.0	0.0	1.0	0.0	0.83	0.24
keywhiz	1.0	0.0	1.0	0.0	1.0	0.0	0.5	0.53
lenskit	0.99	0.0	1.0	0.0	1.0	0.0	0.73	0.1
less4j	0.96	0.02	1.0	0.0	0.98	0.01	0.74	0.12
logback	0.99	0.0	1.0	0.0	0.99	0.0	0.74	0.07
lorsource	0.99	0.0	1.0	0.01	0.99	0.0	0.7	0.09
maven-git-commit-id-plugin	0.96	0.01	0.99	0.01	0.98	0.01	0.76	0.06

metrics	0.96	0.05	0.98	0.01	0.97	0.03	0.81	0.19
mockito	0.99	0.01	1.0	0.0	0.99	0.0	0.78	0.1
mybatis-3	1.0	0.0	1.0	0.0	1.0	0.0	0.65	0.24
nokogiri	0.92	0.04	0.94	0.05	0.93	0.03	0.82	0.06
nutz	0.86	0.03	0.94	0.04	0.9	0.02	0.69	0.05
okhttp	0.91	0.02	0.96	0.04	0.93	0.01	0.7	0.05
onebusaway-android	1.0	0.0	1.0	0.0	1.0	0.0	0.77	0.15
openwayback	1.0	0.0	1.0	0.0	1.0	0.0	0.87	0.14
owner	1.0	0.0	1.0	0.0	1.0	0.0	0.73	0.19
p6spy	0.92	0.05	0.91	0.08	0.91	0.03	0.8	0.06
parceler	0.99	0.0	1.0	0.0	1.0	0.0	0.65	0.13
pdfsam	0.89	0.04	0.96	0.04	0.92	0.03	0.53	0.16
play-authenticate	0.98	0.01	1.0	0.01	0.99	0.01	0.8	0.11
psi-probe	1.0	0.0	1.0	0.0	1.0	0.0	0.91	0.17
pushy	1.0	0.0	1.0	0.0	1.0	0.0	0.6	0.52
querydsl	0.99	0.0	1.0	0.0	0.99	0.0	0.71	0.09
quickml	0.99	0.0	1.0	0.0	0.99	0.0	0.72	0.04
qulice	0.99	0.0	1.0	0.0	0.99	0.0	0.72	0.18
restlet-framework-java	0.85	0.11	0.92	0.07	0.88	0.09	0.71	0.23
retrofit	1.0	0.0	1.0	0.0	1.0	0.0	0.88	0.1
rewrite	0.91	0.05	0.96	0.05	0.93	0.04	0.74	0.15
rexster	1.0	0.0	1.0	0.0	1.0	0.0	0.82	0.1
robospice	0.85	0.05	0.76	0.1	0.8	0.06	0.67	0.08
rultor	0.86	0.06	0.94	0.04	0.9	0.03	0.63	0.13
rxjava-jdbc	1.0	0.0	1.0	0.0	1.0	0.0	0.72	0.29
selendroid	0.95	0.06	0.97	0.04	0.96	0.05	0.86	0.16
seyren	0.99	0.01	1.0	0.0	0.99	0.0	0.51	0.37
sms-backup-plus	1.0	0.0	1.0	0.0	1.0	0.0	0.78	0.09
spark	0.99	0.0	1.0	0.0	1.0	0.0	0.71	0.14
spring-cloud-config	0.98	0.01	1.0	0.0	0.99	0.0	0.72	0.08
springside4	0.86	0.05	0.92	0.04	0.89	0.03	0.65	0.1
storio	0.97	0.0	1.0	0.0	0.99	0.0	0.72	0.06
storm	0.86	0.07	0.8	0.1	0.83	0.06	0.72	0.1
structr	0.82	0.1	0.76	0.13	0.78	0.06	0.63	0.1
stubby4j	0.85	0.03	0.96	0.02	0.9	0.02	0.69	0.07
thredds	0.92	0.03	0.97	0.03	0.94	0.02	0.76	0.09
traccar	1.0	0.0	1.0	0.0	1.0	0.0	0.59	0.15
truth	0.97	0.01	1.0	0.0	0.98	0.01	0.72	0.1
twilio-java	0.9	0.02	0.98	0.01	0.94	0.01	0.62	0.05
u2020	0.99	0.01	0.99	0.03	0.99	0.02	0.74	0.19
unirest-java	0.97	0.01	0.98	0.04	0.97	0.02	0.62	0.17
waffle	1.0	0.0	1.0	0.0	1.0	0.0	0.75	0.11
webcam-capture	1.0	0.0	1.0	0.0	1.0	0.0	0.7	0.05
wire	1.0	0.0	1.0	0.0	1.0	0.0	0.6	0.52
xtreemfs	0.99	0.0	1.0	0.01	0.99	0.0	0.84	0.05
yobi	1.0	0.0	1.0	0.0	1.0	0.0	0.73	0.19

Table 8.19: Descriptive statistics of the dependent variables for an RF model after applying CF.

Project	**Precision**		**Recall**		**F1**		**MCC**	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD

AcDisplay	0.7	0.07	0.48	0.13	0.56	0.08	0.34	0.07
DDT	0.85	0.01	0.98	0.01	0.91	0.01	0.39	0.06
DSpace	0.98	0.0	1.0	0.0	0.99	0.0	0.56	0.1
HearthSim	0.99	0.0	1.0	0.0	1.0	0.0	0.31	0.23
HikariCP	0.91	0.02	0.99	0.01	0.95	0.01	0.45	0.12
Hydra	0.95	0.01	1.0	0.0	0.97	0.0	0.54	0.07
Hystrix	0.75	0.09	0.76	0.17	0.74	0.08	0.41	0.07
Jest	0.92	0.01	0.99	0.01	0.95	0.01	0.48	0.08
LittleProxy	0.92	0.01	0.99	0.01	0.95	0.01	0.49	0.06
MozStumbler	1.0	0.0	1.0	0.0	1.0	0.0	0.44	0.48
OpenRefine	0.97	0.02	1.0	0.01	0.98	0.01	0.76	0.15
ProjectRed	0.9	0.09	0.96	0.06	0.92	0.05	0.8	0.15
RoaringBitmap	0.99	0.0	1.0	0.0	0.99	0.0	0.42	0.08
Singularity	0.82	0.05	0.94	0.08	0.87	0.04	0.45	0.15
Dspace	0.98	0.0	1.0	0.0	0.99	0.0	0.56	0.10
auto	0.99	0.0	1.0	0.0	1.00	0.0	0.59	0.15
airlift	0.77	0.19	0.77	0.14	0.76	0.15	0.61	0.26
analytics-android	0.96	0.01	1.0	0.0	0.98	0.0	0.6	0.1
android	0.87	0.05	0.97	0.06	0.92	0.02	0.7	0.09
android-maven-plugin	0.96	0.01	0.98	0.04	0.97	0.02	0.43	0.09
assertj-android	0.81	0.02	0.98	0.01	0.89	0.01	0.4	0.08
auto	0.99	0.0	1.0	0.0	1.0	0.0	0.59	0.15
basex	0.98	0.0	1.0	0.01	0.99	0.0	0.5	0.13
blueflood	0.92	0.01	1.0	0.0	0.96	0.0	0.51	0.06
blueprints	0.83	0.06	0.9	0.07	0.86	0.03	0.6	0.1
bnd	0.93	0.01	1.0	0.0	0.96	0.0	0.56	0.04
brightspot-cms	0.88	0.04	0.96	0.03	0.92	0.02	0.71	0.07
cas-addons	1.0	0.0	1.0	0.0	1.0	0.0	0.72	0.22
cassandra-reaper	0.91	0.02	0.99	0.01	0.95	0.01	0.53	0.14
ccw	0.85	0.02	0.97	0.02	0.91	0.02	0.6	0.08
checkstyle	1.0	0.0	1.0	0.0	1.0	0.0	0.59	0.12
cloudify	0.86	0.01	0.96	0.07	0.9	0.03	0.32	0.08
core	1.0	0.0	1.0	0.0	1.0	0.0	0.31	0.23
dagger	0.98	0.01	1.0	0.0	0.99	0.0	0.64	0.12
dropwizard	0.99	0.0	1.0	0.0	0.99	0.0	0.53	0.09
dynjs	0.98	0.01	1.0	0.0	0.99	0.0	0.59	0.21
error-prone	1.0	0.0	1.0	0.0	1.0	0.0	0.31	0.41
frontend-maven-plugin	0.97	0.0	0.99	0.02	0.98	0.01	0.48	0.1
go-lang-idea-plugin	0.98	0.01	0.99	0.03	0.98	0.01	0.59	0.15
goclipse	1.0	0.0	1.0	0.0	1.0	0.0	0.5	0.15
gpslogger	0.96	0.01	1.0	0.01	0.98	0.01	0.55	0.13
hivemall	1.0	0.0	1.0	0.0	1.0	0.0	0.61	0.14
htm.java	1.0	0.0	1.0	0.0	1.0	0.0	0.68	0.22
idea-gitignore	0.87	0.01	0.99	0.01	0.92	0.01	0.35	0.14
jInstagram	0.99	0.0	1.0	0.0	0.99	0.0	0.53	0.09
jPOS	1.0	0.0	1.0	0.0	1.0	0.0	0.62	0.15
jade4j	0.99	0.0	1.0	0.0	0.99	0.0	0.35	0.23
javaslang	1.0	0.0	1.0	0.0	1.0	0.0	0.65	0.18
jcabi-aspects	0.9	0.03	0.99	0.01	0.94	0.02	0.53	0.17
jcabi-github	0.86	0.08	0.86	0.16	0.86	0.11	0.61	0.25
jcabi-http	0.91	0.01	0.99	0.02	0.95	0.01	0.55	0.09
jedis	0.98	0.0	0.99	0.02	0.99	0.01	0.5	0.1
jmeter-plugins	0.97	0.03	0.99	0.01	0.98	0.01	0.83	0.12
jmonkeyengine	0.99	0.0	1.0	0.0	1.0	0.0	0.6	0.14
jmxtrans	0.99	0.01	1.0	0.0	0.99	0.0	0.56	0.23
joda-time	1.0	0.0	1.0	0.0	1.0	0.0	0.58	0.36

jodd	0.99	0.01	0.81	0.41	0.81	0.39	0.38	0.24
jphp	0.92	0.02	0.99	0.02	0.95	0.01	0.54	0.12
jsonld-java	1.0	0.0	1.0	0.0	1.0	0.0	0.62	0.17
jsprit	1.0	0.0	1.0	0.0	1.0	0.0	0.74	0.21
keywhiz	1.0	0.0	1.0	0.0	1.0	0.0	0.5	0.53
lenskit	0.99	0.0	1.0	0.0	1.0	0.0	0.55	0.09
less4j	0.95	0.02	1.0	0.0	0.97	0.01	0.64	0.17
logback	0.98	0.0	1.0	0.0	0.99	0.0	0.49	0.08
lorsource	0.99	0.0	1.0	0.0	0.99	0.0	0.52	0.17
maven-git-commit-id-plugin	0.92	0.01	0.99	0.01	0.96	0.01	0.49	0.07
metrics	0.9	0.05	0.98	0.01	0.94	0.03	0.6	0.23
mockito	0.98	0.01	0.99	0.02	0.98	0.01	0.55	0.13
mybatis-3	1.0	0.0	1.0	0.0	1.0	0.0	0.5	0.28
nokogiri	0.84	0.06	0.89	0.04	0.86	0.03	0.65	0.1
nutz	0.78	0.04	0.87	0.07	0.82	0.03	0.43	0.09
okhttp	0.84	0.01	0.95	0.06	0.89	0.03	0.49	0.09
onebusaway-android	1.0	0.0	1.0	0.0	1.0	0.0	0.57	0.14
openwayback	1.0	0.0	1.0	0.0	1.0	0.0	0.7	0.19
owner	0.99	0.0	1.0	0.0	1.0	0.0	0.44	0.19
p6spy	0.72	0.12	0.92	0.12	0.8	0.07	0.39	0.29
parceler	0.99	0.0	1.0	0.0	0.99	0.0	0.49	0.16
pdfsam	0.85	0.05	0.94	0.13	0.89	0.05	0.28	0.1
play-authenticate	0.97	0.01	1.0	0.01	0.98	0.01	0.64	0.14
psi-probe	1.0	0.0	1.0	0.0	1.0	0.0	0.81	0.26
pushy	1.0	0.0	1.0	0.0	1.0	0.0	0.2	0.42
querydsl	0.98	0.0	1.0	0.0	0.99	0.0	0.59	0.08
quickml	0.98	0.0	1.0	0.0	0.99	0.0	0.51	0.12
qulice	0.99	0.0	1.0	0.0	0.99	0.0	0.61	0.17
restlet-framework-java	0.8	0.13	0.64	0.25	0.67	0.14	0.42	0.17
retrofit	0.99	0.0	1.0	0.0	0.99	0.0	0.62	0.11
rewrite	0.82	0.04	0.89	0.12	0.85	0.05	0.43	0.13
rexster	0.99	0.0	1.0	0.0	1.0	0.0	0.61	0.15
robospice	0.69	0.07	0.62	0.14	0.64	0.09	0.41	0.12
rultor	0.8	0.06	0.89	0.09	0.83	0.04	0.41	0.14
rxjava-jdbc	1.0	0.0	1.0	0.0	1.0	0.0	0.52	0.37
selendroid	0.82	0.06	0.95	0.02	0.88	0.03	0.55	0.14
seyren	0.98	0.01	1.0	0.0	0.99	0.0	0.27	0.32
sms-backup-plus	0.99	0.0	1.0	0.0	1.0	0.0	0.66	0.18
spark	0.99	0.0	1.0	0.0	0.99	0.0	0.42	0.22
spring-cloud-config	0.96	0.01	1.0	0.0	0.98	0.0	0.51	0.15
springside4	0.78	0.04	0.83	0.19	0.79	0.11	0.42	0.12
storio	0.96	0.01	1.0	0.0	0.98	0.0	0.47	0.09
storm	0.73	0.1	0.67	0.19	0.68	0.11	0.49	0.14
structr	0.65	0.12	0.66	0.23	0.62	0.1	0.36	0.1
stubby4j	0.77	0.03	0.92	0.08	0.83	0.03	0.47	0.08
thredds	0.85	0.03	0.96	0.03	0.9	0.02	0.56	0.11
traccar	1.0	0.0	1.0	0.0	1.0	0.0	0.41	0.18
truth	0.95	0.01	1.0	0.0	0.98	0.0	0.57	0.1
twilio-java	0.84	0.01	0.99	0.01	0.91	0.01	0.38	0.06
u2020	0.98	0.01	1.0	0.0	0.99	0.0	0.48	0.21
unirest-java	0.95	0.01	1.0	0.0	0.98	0.01	0.49	0.15
waffle	0.99	0.0	1.0	0.0	1.0	0.0	0.45	0.22
webcam-capture	1.0	0.0	1.0	0.0	1.0	0.0	0.56	0.09
wire	1.0	0.0	1.0	0.0	1.0	0.0	0.6	0.52
xtreemfs	0.98	0.01	0.99	0.01	0.99	0.0	0.69	0.14
yobi	1.0	0.0	1.0	0.0	1.0	0.0	0.62	0.16

Table 8.20: The distribution of build outcomes within the analyzed subjects.

project	**class0**	**class1**	**class_ratio**
AcDisplay	19424	15461	44.319
DDT	11929	51879	18.695
DSpace	2330	83544	2.713
HearthSim	400	66046	0.601
HikariCP	3952	28938	12.015
Hydra	623	7270	7.893
Hystrix	18558	24944	42.66
Jest	2862	22863	11.125
LittleProxy	1025	7737	11.698
MozStumbler	15	8625	0.173
OpenRefine	357	3814	8.559
ProjectRed	450	702	39.062
RoaringBitmap	788	41989	1.842
Singularity	2267	6257	26.595
airlift	14632	9380	39.063
analytics-android	507	6926	6.820
android	4587	11046	29.341
android-maven-plugin	4598	80031	5.433
assertj-android	3798	12111	23.873
auto	137	12917	1.049
basex	2030	57303	3.421
blueflood	4886	40437	10.78
blueprints	16890	29439	36.456
bnd	3852	31838	10.7929
brightspot-cms	2981	7040	29.747
cas-addons	66	8049	0.813
cassandra-reaper	1082	7324	12.871
ccw	4070	11538	26.076
checkstyle	370	103059	0.357
cloudify	56428	265933	17.504
core	420	84118	0.496
dagger	119	3355	3.425
dropwizard	1154	54038	2.090
dynjs	1033	35286	2.844
error-prone	26	101295	0.025
frontend-maven-plugin	110	2774	3.814
go-lang-idea-plugin	1319	32505	3.899
goclipse	509	76552	0.66
gpslogger	1009	15405	6.147
hivemall	169	22752	0.737
htm.java	470	53930	0.863
idea-gitignore	4641	24359	16.003
jInstagram	294	20225	1.432
jPOS	160	38231	0.416
jade4j	283	16171	1.719

java-design-patterns	55	23249	0.236
javaslang	1151	193409	0.591
jcabi-aspects	904	4669	16.221
jcabi-github	5937	11839	33.399
jcabi-http	624	3913	13.753
jedis	984	40365	2.379
jinjava	8	12747	0.062
jmeter-plugins	8995	56751	13.681
jmonkeyengine	928	77326	1.185
jmxtrans	198	8497	2.277
joda-time	37	19276	0.191
jodd	1546	153410	0.997
jphp	20586	141821	12.675
jsonld-java	78	34127	0.228
jsonschema2pojo	3	15091	0.0198
jsprit	118	23640	0.496
keywhiz	5	12791	0.039
lenskit	848	58210	1.435
less4j	8259	73837	10.060
logback	2046	71501	2.781
lorsource	751	34964	2.102
maven-git-commit-id-plugin	1714	14099	10.839
metrics	1778	7857	18.453
mockito	2976	71665	3.987
mybatis-3	604	105989	0.566
nodeclipse-1	27	23959	0.112
nokogiri	10368	15481	40.109
nutz	22390	42002	34.771
okhttp	17121	53828	24.1314
onebusaway-android	66	22335	0.294
openwayback	31	12905	0.239
owner	266	23163	1.135
p6spy	6353	8975	41.447
parceler	284	17457	1.6
pdfsam	24027	106294	18.436
picard	378	11017	3.317
play-authenticate	330	5229	5.936
psi-probe	100	57045	0.174
pushy	9	4705	0.19
querydsl	929	38383	2.363
quickml	486	14942	3.15
qulice	253	11216	2.205
restlet-framework-java	52947	65339	44.761
retrofit	358	18901	1.858
rewrite	4091	10717	27.627
rexster	515	37728	1.346
robospice	8475	6487	43.356
rultor	10632	23676	30.989
rxjava-jdbc	20	8677	0.229
selendroid	18702	42747	30.435

seyren	166	8447	1.927
sms-backup-plus	215	16358	1.297
spark	94	6345	1.459
spring-cloud-config	1368	23983	5.396
springside4	7317	14330	33.801
storio	911	14309	5.985
storm	18316	12936	41.392
structr	73595	57331	43.788
stubby4j	14505	27490	34.539
thredds	6920	20015	25.691
traccar	144	76165	0.188
truth	1351	17931	7.006
twilio-java	6285	25939	19.504
u2020	149	4729	3.0545
unirest-java	262	3803	6.445
waffle	168	20320	0.8199
webcam-capture	267	38456	0.689
wire	11	52079	0.0211
xtreemfs	2494	53452	4.457
yobi	106	23134	0.4561

Bibliography

[1] A. Brand, L. Allen, M. Altman, M. Hlava and J. Scott, 'Beyond authorship: Attribution, contribution, collaboration, and credit,' *Learned Publishing*, vol. 28, no. 2, pp. 151–155, 2015 (cit. on p. vii).

[2] I.-C. Donca, O. P. Stan, M. Misaros, D. Gota and L. Miclea, 'Method for continuous integration and deployment using a pipeline generator for agile software projects,' *Sensors*, vol. 22, no. 12, p. 4637, 2022 (cit. on pp. 1, 155).

[3] A. E. Hassan and K. Zhang, 'Using decision trees to predict the certification result of a build,' in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, IEEE, 2006, pp. 189–198 (cit. on pp. 1, 137).

[4] J. Xia and Y. Li, 'Could we predict the result of a continuous integration build? an empirical study,' in *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, IEEE, 2017, pp. 311–315 (cit. on pp. 2, 137, 139).

[5] Z. Nazari, M. Nazari, M Sayed and S Danish, 'Evaluation of class noise impact on performance of machine learning algorithms,' *IJCSNS Int. J. Comput. Sci. Netw. Secur*, vol. 18, p. 149, 2018 (cit. on p. 2).

[6] S. Gupta and A. Gupta, 'Dealing with noise problem in machine learning data-sets: A systematic review,' *Procedia Computer Science*, vol. 161, pp. 466–474, 2019 (cit. on pp. 2, 155).

[7] G. A. Liebchen, 'Data cleaning techniques for software engineering data sets,' Ph.D. dissertation, Brunel University, School of Information Systems, Computing and Mathematics, 2010 (cit. on pp. 2, 14, 69, 155).

[8] I. Saidani, A. Ouni and M. W. Mkaouer, 'Improving the prediction of continuous integration build failures using deep learning,' *Automated Software Engineering*, vol. 29, no. 1, p. 21, 2022 (cit. on pp. 4, 12).

[9] S. Arachchi and I. Perera, 'Continuous integration and continuous delivery pipeline automation for agile software project management,' in *2018 Moratuwa Engineering Research Conference (MERCon)*, IEEE, 2018, pp. 156–161 (cit. on p. 4).

[10] T. Yu and T. Wang, ‘A study of regression test selection in continuous integration environments,’ in *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 2018, pp. 135–143 (cit. on p. 5).

[11] S. García, J. Luengo and F. Herrera, ‘Dealing with noisy data,’ in *Data Preprocessing in Data Mining.* Cham: Springer International Publishing, 2015, pp. 107–145, ISBN: 978-3-319-10247-4. DOI: `10.1007/978-3-319-10247-4_5` (cit. on p. 5).

[12] A. Ahmad, F. G. de Oliveira Neto, Z. Shi, K. Sandahl and O. Leifler, ‘A multi-factor approach for flaky test detection and automated root cause analysis,’ in *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*, IEEE, 2021, pp. 338–348 (cit. on p. 7).

[13] J. D. Van Hulse, T. M. Khoshgoftaar and H. Huang, ‘The pairwise attribute noise detection algorithm,’ *Knowledge and Information Systems*, vol. 11, no. 2, pp. 171–190, 2007 (cit. on pp. 8, 9, 16, 23, 69, 72, 78, 81, 95, 96).

[14] D. Guan, W. Yuan, Y.-K. Lee and S. Lee, ‘Identifying mislabeled training data with the aid of unlabeled data,’ *Applied Intelligence*, vol. 35, no. 3, pp. 345–358, 2011 (cit. on pp. 9, 13, 67, 68, 71, 158).

[15] C. E. Brodley, M. A. Friedl *et al.*, ‘Identifying and eliminating mislabeled training instances,’ in *Proceedings of the National Conference on Artificial Intelligence*, 1996, pp. 799–805 (cit. on pp. 9, 13, 68, 71, 72).

[16] T. M. Khoshgoftaar and J. Van Hulse, ‘Identifying noise in an attribute of interest,’ in *Fourth International Conference on Machine Learning and Applications (ICMLA’05)*, IEEE, 2005, 6–pp (cit. on pp. 9, 68, 72).

[17] K.-A. Yoon and D.-H. Bae, ‘A pattern-based outlier detection method identifying abnormal attributes in software project data,’ *Information and Software Technology*, vol. 52, no. 2, pp. 137 –151, 2010, ISSN: 0950-5849. DOI: `https://doi.org/10.1016/j.infsof.2009.08.005` (cit. on pp. 9, 68).

[18] D. Gamberger, N. Lavrac and S. Dzeroski, ‘Noise detection and elimination in data preprocessing: Experiments in medical domains,’ *Applied artificial intelligence*, vol. 14, no. 2, pp. 205–223, 2000 (cit. on pp. 9, 51).

[19] D. Gamberger and N. Lavrač, ‘Conditions for occam’s razor applicability and noise elimination,’ in *European Conference on Machine Learning*, Springer, 1997, pp. 108–123 (cit. on p. 9).

[20] C.-M. Teng, ‘Correcting noisy data.,’ in *ICML*, Citeseer, 1999, pp. 239–248 (cit. on pp. 9, 158, 187).

[21] C. E. Brodley and M. A. Friedl, ‘Identifying mislabeled training data,’ *Journal of artificial intelligence research*, vol. 11, pp. 131–167, 1999 (cit. on pp. 10, 11, 33, 73, 94, 157, 163).

[22] M. Samami, E. Akbari, M. Abdar *et al.*, 'A mixed solution-based high agreement filtering method for class noise detection in binary classification,' *Physica A: Statistical Mechanics and its Applications*, vol. 553, p. 124 219, 2020 (cit. on pp. 10, 34).

[23] T. M. Khoshgoftaar, V. Joshi and N. Seliya, 'Detecting noisy instances with the ensemble filter: A study in software quality estimation,' *International Journal of Software Engineering and Knowledge Engineering*, vol. 16, no. 01, pp. 53–76, 2006 (cit. on p. 10).

[24] P. Rebours and T. M. Khoshgoftaar, 'Quality problem in software measurement data,' in *Advances in Computers*, vol. 66, Elsevier, 2006, pp. 43–77 (cit. on pp. 11, 160, 163).

[25] E. Giger, M. D'Ambros, M. Pinzger and H. C. Gall, 'Method-level bug prediction,' in *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, 2012, pp. 171–180 (cit. on p. 11).

[26] A. Perera, A. Aleti, B. Turhan and M. Böhme, 'An experimental assessment of using theoretical defect predictors to guide search-based software testing,' *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 131–146, 2022 (cit. on p. 11).

[27] B. Caglayan, B. Turhan, A. Bener, M. Habayeb, A. Miransky and E. Cialini, 'Merits of organizational metrics in defect prediction: An industrial replication,' in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, IEEE, vol. 2, 2015, pp. 89–98 (cit. on p. 11).

[28] V. R. Basili, L. C. Briand and W. L. Melo, 'A validation of object-oriented design metrics as quality indicators,' *IEEE Transactions on software engineering*, vol. 22, no. 10, pp. 751–761, 1996 (cit. on p. 12).

[29] A. Bernstein, J. Ekanayake and M. Pinzger, 'Improving defect prediction using temporal features and non linear models,' in *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, 2007, pp. 11–18 (cit. on p. 12).

[30] A. E. Hassan, 'Predicting faults using the complexity of code changes,' in *2009 IEEE 31st international conference on software engineering*, IEEE, 2009, pp. 78–88 (cit. on pp. 12, 13).

[31] E. Arisholm and L. C. Briand, 'Predicting fault-prone components in a java legacy system,' in *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, 2006, pp. 8–17 (cit. on p. 12).

[32] A. Groce, T. Kulesza, C. Zhang *et al.*, 'You are the only possible oracle: Effective test selection for end users of interactive machine learning systems,' *IEEE Transactions on Software Engineering*, vol. 40, no. 3, pp. 307–323, 2013 (cit. on p. 12).

[33] T. Zimmermann, N. Nagappan, H. Gall, E. Giger and B. Murphy, 'Cross-project defect prediction: A large scale experiment on data vs. domain vs. process,' in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2009, pp. 91–100 (cit. on p. 12).

[34] B. Chen, L. Chen, C. Zhang and X. Peng, 'Buildfast: History-aware build outcome prediction for fast feedback and reduced cost in continuous integration,' in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 42–53 (cit. on p. 12).

[35] L. Zhang, B. Cui and Z. Zhang, 'Optimizing continuous integration by dynamic test proportion selection,' in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2023, pp. 438–449 (cit. on p. 12).

[36] J. Van Hulse and T. Khoshgoftaar, 'Knowledge discovery from imbalanced and noisy data,' *Data & Knowledge Engineering*, vol. 68, no. 12, pp. 1513–1542, 2009 (cit. on p. 13).

[37] A. Folleco, T. M. Khoshgoftaar, J. Van Hulse and L. Bullard, 'Software quality modeling: The impact of class noise on the random forest classifier,' in *2008 IEEE congress on evolutionary computation (IEEE world congress on computational intelligence)*, IEEE, 2008, pp. 3853–3859 (cit. on p. 13).

[38] T. M. Khoshgoftaar and N. Seliya, 'The necessity of assuring quality in software measurement data,' in *10th International Symposium on Software Metrics, 2004. Proceedings.*, IEEE, 2004, pp. 119–130 (cit. on p. 13).

[39] T. M. Khoshgoftaar and P. Rebours, 'Improving software quality prediction by noise filtering techniques,' *Journal of Computer Science and Technology*, vol. 22, no. 3, pp. 387–396, 2007 (cit. on pp. 13, 155).

[40] F. Muhlenbach, S. Lallich and D. A. Zighed, 'Identifying and handling mislabelled instances,' *Journal of Intelligent Information Systems*, vol. 22, no. 1, pp. 89–109, 2004, ISSN: 1573-7675. DOI: `10.1023/A:1025832930864`. [Online]. Available: `https://doi.org/10.1023/A:1025832930864` (cit. on pp. 13, 67, 72, 79, 158).

[41] T. M. Khoshgoftaar, N. Seliya and K. Gao, 'Rule-based noise detection for software measurement data,' in *Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration, 2004. IRI 2004.*, IEEE, 2004, pp. 302–307 (cit. on pp. 14, 67, 72).

[42] M. Ochodek, M. Staron, D. Bargowski, W. Meding and R. Hebig, 'Using machine learning to design a flexible loc counter,' in *2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, IEEE, 2017, pp. 14–20 (cit. on pp. 15, 38, 40, 56, 74, 129, 142, 170).

[43] I. Saidani, A. Ouni, M. Chouchen and M. W. Mkaouer, 'Predicting continuous integration build failures using evolutionary search,' *Information and Software Technology*, vol. 128, p. 106 392, 2020 (cit. on p. 17).

[44] L. Zhang, J.-H. Tian, J. Jiang, Y.-J. Liu, M.-Y. Pu and T. Yue, 'Empirical research in software engineering—a literature survey,' *Journal of Computer Science and Technology*, vol. 33, pp. 876–899, 2018 (cit. on p. 19).

[45] P. Runeson, E. Engström and M.-A. Storey, 'The design science paradigm as a frame for empirical software engineering,' in *Contemporary empirical methods in software engineering*, Springer, 2020, pp. 127–147 (cit. on p. 19).

[46] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell and A. Wesslén, *Experimentation in software engineering.* Springer Science & Business Media, 2012 (cit. on pp. 19, 20, 30, 48, 63, 95, 118, 133, 152, 189).

[47] S. Yoo and M. Harman, 'Regression testing minimization, selection and prioritization: A survey,' *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012 (cit. on p. 21).

[48] M. Usman, R. Britto, J. Börstler and E. Mendes, 'Taxonomies in software engineering: A systematic mapping study and a revised taxonomy development method,' *Information and Software Technology*, vol. 85, pp. 43–59, 2017 (cit. on pp. 24, 101, 104).

[49] M. Beller, G. Gousios and A. Zaidman, 'Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration,' in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, IEEE, 2017, pp. 447–450 (cit. on pp. 25, 138, 140, 143).

[50] S. Yatish, J. Jiarpakdee, P. Thongtanunam and C. Tantithamthavorn, 'Mining software defects: Should we consider affected releases?' In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE, 2019, pp. 654–665 (cit. on p. 32).

[51] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara and K. Matsumoto, 'The impact of mislabelling on the performance and interpretation of defect prediction models,' in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, IEEE, vol. 1, 2015, pp. 812–823 (cit. on p. 32).

[52] B. Sluban and N. Lavrač, 'Relating ensemble diversity and performance: A study in class noise detection,' *Neurocomputing*, vol. 160, pp. 120–131, 2015 (cit. on pp. 32, 34, 52, 161).

[53] Y. Ma and H. He, 'Imbalanced learning: Foundations, algorithms, and applications,' 2013 (cit. on p. 33).

[54] X. Zhu and X. Wu, 'Class noise vs. attribute noise: A quantitative study,' *Artificial intelligence review*, vol. 22, no. 3, pp. 177–210, 2004 (cit. on pp. 33, 51, 67, 68, 73, 79, 94, 155).

[55] 'Iso/iec/ieee international standard - software and systems engineering –software testing–part 1: Concepts and definitions,' Tech. Rep., 2020, pp. 1–50 (cit. on pp. 35, 102, 105, 109, 121, 130).

[56] D. Ståhl and J. Bosch, 'Experienced benefits of continuous integration in industry software product development: A case study,' in *The 12th IASTED International Conference on Software Engineering,(Innsbruck, Austria, 2013)*, 2013, pp. 736–743 (cit. on p. 37).

[57] G. Çalikli, M. Staron and W. Meding, 'Measure early and decide fast: Transforming quality management and measurement to continuous deployment,' in *Proceedings of the 2018 International Conference on Software and System Process*, ACM, 2018, pp. 51–60 (cit. on p. 37).

[58] E. Knauss, M. Staron, W. Meding, O. Söder, A. Nilsson and M. Castell, 'Supporting continuous integration by code-churn based test selection,' in *Proceedings of the Second International Workshop on Rapid Continuous Software Engineering*, IEEE Press, 2015, pp. 19–25 (cit. on pp. 37, 40, 99, 121).

[59] N. Nagappan and T. Ball, 'Use of relative code churn measures to predict system defect density,' in *Proceedings of the 27th international conference on Software engineering*, ACM, 2005, pp. 284–292 (cit. on pp. 37, 39, 42, 67).

[60] F. Chollet, *Deep Learning with Python.* Manning, 2017 (cit. on pp. 38, 39).

[61] A. Géron, *Hands-On Machine Learning with Scikit-Learn and Tensor-Flow.* Oreilly, 2015 (cit. on pp. 38, 39).

[62] R. Saxena, *Introduction to decision tree algorithm*, 2017. [Online]. Available: `https://dataaspirant.com/2017/01/30/how-decision-tree-algorithm-works/` (visited on 24/04/2019) (cit. on p. 38).

[63] M. Awad and R. Khanna, *Efficient learning machines: theories, concepts, and applications for engineers and system designers.* Apress, 2017 (cit. on p. 38).

[64] I. Gondra, 'Applying machine learning to software fault-proneness prediction,' *Journal of Systems and Software*, vol. 81, no. 2, pp. 186–195, 2008 (cit. on p. 39).

[65] Y. Shin, A. Meneely, L. Williams and J. A. Osborne, 'Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities,' *IEEE Transactions on Software Engineering*, vol. 37, no. 6, pp. 772–787, 2011, ISSN: 0098-5589. DOI: `10.1109/TSE.2010.81` (cit. on p. 39).

[66] T. L. Graves, A. F. Karr, J. S. Marron and H. Siy, 'Predicting fault incidence using software change history,' *IEEE Transactions on Software Engineering*, vol. 26, no. 7, pp. 653–661, 2000, ISSN: 0098-5589. DOI: `10.1109/32.859533` (cit. on p. 39).

[67] J. Beningo, *Using the static keyword in c*, `https://community.arm.com/developer/ip-products/system/b/embedded-blog/posts/using-the-static-keyword-in-c`, 2014 (cit. on p. 39).

[68] V. H. Durelli, R. S. Durelli, S. S. Borges *et al.*, 'Machine learning applied to software testing: A systematic mapping study,' *IEEE Transactions on Reliability*, 2019 (cit. on pp. 39, 122).

[69] B. Busjaeger and T. Xie, 'Learning for test prioritization: An industrial case study,' in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM, 2016, pp. 975–980 (cit. on p. 39).

[70] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang and B. Xie, 'Learning to prioritize test programs for compiler testing,' in *Proceedings of the 39th International Conference on Software Engineering*, IEEE Press, 2017, pp. 700–711 (cit. on p. 40).

[71] H. Spieker, A. Gotlieb, D. Marijan and M. Mossige, 'Reinforcement learning for automatic test case prioritization and selection in continuous integration,' in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ACM, 2017, pp. 12–22 (cit. on p. 40).

[72] M. Azizi and H. Do, 'A collaborative filtering recommender system for test case prioritization in web applications,' in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, ser. SAC '18, Pau, France: ACM, 2018, pp. 1560–1567, ISBN: 978-1-4503-5191-1. DOI: `10.1145/3167132.3167299`. [Online]. Available: `http://doi.acm.org/10.1145/3167132.3167299` (cit. on p. 40).

[73] F. Palma, T. Abdou, A. Bener, J. Maidens and S. Liu, 'An improvement to test case failure prediction in the context of test case prioritization,' in *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*, ser. PROMISE'18, Oulu, Finland: ACM, 2018, pp. 80–89, ISBN: 978-1-4503-6593-2. DOI: `10.1145/3273934.3273944`. [Online]. Available: `http://doi.acm.org/10.1145/3273934.3273944` (cit. on p. 40).

[74] T. B. Noor and H. Hemmati, 'Studying test case failure prediction for test case prioritization,' in *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, ACM, 2017, pp. 2–11 (cit. on pp. 40, 67).

[75] T. B. Noor and H. Hemmati, 'A similarity-based approach for test case prioritization using historical failure data,' in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 2015, pp. 58–68 (cit. on p. 40).

[76] F. Pedregosa, G. Varoquaux, A. Gramfort *et al.*, 'Scikit-learn: Machine learning in Python,' *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011 (cit. on pp. 46, 57, 58, 84, 86, 129, 146, 171, 172).

[77] F. Chollet *et al.*, *Keras*, `https://keras.io`, 2015 (cit. on pp. 46, 171).

[78] K. W. Al-Sabbagh, M. Staron, R. Hebig and W. Meding, ‘Predicting test case verdicts using textual analysis of committed code churns,’ in *Joint Proceedings of the International Workshop on Software Measurementand the International Conference on Software Process and Product Measurement (IWSM Mensura 2019)*, vol. 2476, 2019, pp. 138–153 (cit. on pp. 51, 55, 67, 70, 74, 99, 121, 129, 145).

[79] H. Hata, O. Mizuno and T. Kikuno, ‘Fault-prone module detection using large-scale text features based on spam filtering,’ *Empirical Software Engineering*, vol. 15, no. 2, pp. 147–165, 2010 (cit. on pp. 51, 55, 70).

[80] S. Kim, E. J. Whitehead Jr and Y. Zhang, ‘Classifying software changes: Clean or buggy?’ *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 181–196, 2008 (cit. on pp. 51, 55, 67, 70, 71).

[81] L. Aversano, L. Cerulo and C. Del Grosso, ‘Learning from bug-introducing changes to prevent fault prone code,’ in *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, ACM, 2007, pp. 19–26 (cit. on pp. 51, 70, 71).

[82] G. H. John, ‘Robust decision trees: Removing outliers from databases.,’ in *KDD*, vol. 95, 1995, pp. 174–179 (cit. on p. 51).

[83] Q. Zhao and T. Nishida, ‘Using qualitative hypotheses to identify inaccurate data,’ *Journal of Artificial Intelligence Research*, vol. 3, pp. 119–145, 1995 (cit. on p. 51).

[84] J. A. Sáez, J. Luengo and F. Herrera, ‘Evaluating the classifier behavior with noisy data considering performance and robustness: The equalized loss of accuracy measure,’ *Neurocomputing*, vol. 176, pp. 26–35, 2016 (cit. on pp. 51, 79).

[85] D. Guan, W. Yuan and L. Shen, ‘Class noise detection by multiple voting,’ in *2013 Ninth International Conference on Natural Computation (ICNC)*, IEEE, 2013, pp. 906–911 (cit. on p. 52).

[86] D. F. Nettleton, A. Orriols-Puig and A. Fornells, ‘A study of the effect of different types of noise on the precision of supervised learning techniques,’ *Artificial intelligence review*, vol. 33, no. 4, pp. 275–306, 2010 (cit. on p. 54).

[87] J. Zhang and Y. Yang, ‘Robustness of regularized linear classification methods in text categorization,’ in *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, 2003, pp. 190–197 (cit. on p. 54).

[88] J. Abellán and A. R. Masegosa, ‘Bagging decision trees on data sets with classification noise,’ in *International Symposium on Foundations of Information and Knowledge Systems*, Springer, 2010, pp. 248–265 (cit. on p. 54).

[89] M. Pechenizkiy, A. Tsymbal, S. Puuronen and O. Pechenizkiy, ‘Class noise and supervised learning in medical domains: The effect of feature extraction,’ in *19th IEEE symposium on computer-based medical systems (CBMS’06)*, IEEE, 2006, pp. 708–713 (cit. on p. 54).

[90] O. Mizuno, S. Ikami, S. Nakaichi and T. Kikuno, 'Spam filter based approach for finding fault-prone software modules,' in *Proceedings of the Fourth International Workshop on Mining Software Repositories*, IEEE Computer Society, 2007, p. 4 (cit. on pp. 55, 70, 71).

[91] S. Boughorbel, F. Jarray and M. El-Anbari, 'Optimal classifier for imbalanced data using matthews correlation coefficient metric,' *PloS one*, vol. 12, no. 6, 2017 (cit. on p. 56).

[92] B. Frénay and M. Verleysen, 'Classification in the presence of label noise: A survey,' *IEEE transactions on neural networks and learning systems*, vol. 25, no. 5, pp. 845–869, 2013 (cit. on p. 58).

[93] E. Knauss, S. Houmb, K. Schneider, S. Islam and J. Jürjens, 'Supporting requirements engineers in recognising security issues,' in *International Working Conference on Requirements Engineering: Foundation for Software Quality*, Springer, 2011, pp. 4–18 (cit. on p. 67).

[94] M. Ochodek, R. Hebig, W. Meding, G. Frost and M. Staron, 'Recognizing lines of code violating company-specific coding guidelines using machine learning,' *Empirical Software Engineering*, vol. 25, no. 1, pp. 220–265, 2020 (cit. on p. 67).

[95] H. Sajnani, 'Automatic software architecture recovery: A machine learning approach,' in *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, IEEE, 2012, pp. 265–268 (cit. on p. 67).

[96] S. Wang, T. Liu and L. Tan, 'Automatically learning semantic features for defect prediction,' in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, IEEE, 2016, pp. 297–308 (cit. on pp. 67, 75).

[97] Z. Cai, L. Lu and S. Qiu, 'An abstract syntax tree encoding method for cross-project defect prediction,' *IEEE Access*, vol. 7, pp. 170 844–170 853, 2019 (cit. on pp. 67, 75).

[98] K. W. Al-Sabbagh, M. Staron, R. Hebig and W. Meding, 'Improving data quality for regression test selection by reducing annotation noise,' in *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, IEEE, 2020, pp. 191–194 (cit. on pp. 67, 69, 79, 82, 140, 142, 158, 164, 168, 170, 171, 191).

[99] T. Zimmermann and P. Weißgerber, 'Preprocessing cvs data for fine-grained analysis.,' in *MSR*, vol. 4, 2004, pp. 2–6 (cit. on pp. 67, 80).

[100] C. M. Teng, 'Combining noise correction with feature selection,' in *International Conference on Data Warehousing and Knowledge Discovery*, Springer, 2003, pp. 340–349 (cit. on p. 68).

[101] K. W. Al-Sabbagh, R. Hebig and M. Staron, 'The effect of class noise on continuous test case selection: A controlled experiment on industrial data,' in *International Conference on Product-Focused Software Process Improvement*, Springer, 2020, pp. 287–303 (cit. on pp. 71, 168).

[102] T. M. Khoshgoftaar and J. Van Hulse, 'Empirical case studies in attribute noise detection,' *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 39, no. 4, pp. 379–388, 2009 (cit. on p. 72).

[103] C.-M. Teng, 'A comparison of noise handling techniques.,' in *FLAIRS Conference*, 2001, pp. 269–273 (cit. on p. 72).

[104] J. R. Quinlan, 'Induction of decision trees,' *Machine learning*, vol. 1, no. 1, pp. 81–106, 1986 (cit. on p. 73).

[105] R. Moser, W. Pedrycz and G. Succi, 'A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction,' in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 181–190 (cit. on p. 75).

[106] S. Amasaki, Y. Takagi, O. Mizuno and T. Kikuno, 'A bayesian belief network for assessing the likelihood of fault content,' in *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.*, IEEE, 2003, pp. 215–226 (cit. on p. 75).

[107] J. Deng, L. Lu, S. Qiu and Y. Ou, 'A suitable ast node granularity and multi-kernel transfer convolutional neural network for cross-project defect prediction,' *IEEE Access*, vol. 8, pp. 66 647–66 661, 2020 (cit. on p. 75).

[108] T. B. C. Arias, P. Avgeriou and P. America, 'Analyzing the actual execution of a large software-intensive system for determining dependencies,' in *2008 15th Working Conference on Reverse Engineering*, IEEE, 2008, pp. 49–58 (cit. on p. 75).

[109] A. Hamou-Lhadj and T. C. Lethbridge, 'A survey of trace exploration tools and techniques,' in *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, 2004, pp. 42–55 (cit. on p. 75).

[110] M. Balint, R. Marinescu and T. Girba, 'How developers copy,' in *14th IEEE International Conference on Program Comprehension (ICPC'06)*, IEEE, 2006, pp. 56–68 (cit. on p. 80).

[111] V. Ganganwar, 'An overview of classification algorithms for imbalanced datasets,' *International Journal of Emerging Technology and Advanced Engineering*, vol. 2, no. 4, pp. 42–47, 2012 (cit. on p. 84).

[112] A. Axelrod, *Complete Guide to Test Automation.* Springer, 2018 (cit. on p. 99).

[113] K. Wang, C. Zhu, A. Celik, J. Kim, D. Batory and M. Gligoric, 'Towards refactoring-aware regression test selection,' in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, IEEE, 2018, pp. 233–244 (cit. on pp. 99, 121).

[114] B. H. Kwasnik, 'The role of classification in knowledge representation and discovery,' 1999 (cit. on p. 99).

[115] R. Chillarege, I. S. Bhandari, J. K. Chaar *et al.*, 'Orthogonal defect classification-a concept for in-process measurements,' *IEEE Transactions on software Engineering*, vol. 18, no. 11, pp. 943–956, 1992 (cit. on p. 100).

[116] N. Li, Z. Li and X. Sun, 'Classification of software defect detected by black-box testing: An empirical study,' in *2010 Second World Congress on Software Engineering*, IEEE, vol. 2, 2010, pp. 234–240 (cit. on p. 100).

[117] L. Ma and J. Tian, 'Web error classification and analysis for reliability improvement,' *Journal of Systems and Software*, vol. 80, no. 6, pp. 795–804, 2007 (cit. on p. 100).

[118] R. Britto, 'Knowledge classification for supporting effort estimation in global software engineering projects,' Ph.D. dissertation, Blekinge Tekniska Högskola, 2015 (cit. on p. 100).

[119] J. Novak, A. Krajnc *et al.*, 'Taxonomy of static code analysis tools,' in *The 33rd International Convention MIPRO*, IEEE, 2010, pp. 418–422 (cit. on p. 100).

[120] S. Vegas, N. Juristo and V. R. Basili, 'Maturing software engineering knowledge through classifications: A case study on unit testing techniques,' *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 551–565, 2009 (cit. on p. 101).

[121] M. Felderer and I. Schieferdecker, 'A taxonomy of risk-based testing,' *arXiv preprint arXiv:1912.11519*, 2019 (cit. on p. 101).

[122] Y. Liu, C. Xu and S.-C. Cheung, 'Characterizing and detecting performance bugs for smartphone applications,' in *Proceedings of the 36th international conference on software engineering*, 2014, pp. 1013–1024 (cit. on p. 105).

[123] Z. M. Jiang, A. E. Hassan, G. Hamann and P. Flora, 'Automatic identification of load testing problems,' in *2008 IEEE International Conference on Software Maintenance*, IEEE, 2008, pp. 307–316 (cit. on p. 105).

[124] F. Cohen, 'Information system attacks: A preliminary classification scheme,' *Computers & Security*, vol. 16, no. 1, pp. 29–46, 1997 (cit. on pp. 105, 108).

[125] R. C. Seacord and A. D. Householder, 'A structured approach to classifying security vulnerabilities,' CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, Tech. Rep., 2005 (cit. on p. 105).

[126] T. Karttunen, 'Implementing soak testing for an access network solution,' Ph.D. dissertation, HELSINKI UNIVERSITY OF TECHNOLOGY, 2009 (cit. on p. 105).

[127] J. Zhang, S.-C. Cheung and S. T. Chanson, 'Stress testing of distributed multimedia software systems,' in *Formal Methods for Protocol Engineering and Distributed Systems*, Springer, 1999, pp. 119–133 (cit. on p. 105).

[128] D. Cotroneo, R. Pietrantuono, L. Mariani and F. Pastore, 'Investigation of failure causes in workload-driven reliability testing,' in *Fourth international workshop on Software quality assurance: in conjunction with the 6th ESEC/FSE joint meeting*, 2007, pp. 78–85 (cit. on p. 105).

[129] A. Nistor, P.-C. Chang, C. Radoi and S. Lu, 'Caramel: Detecting and fixing performance problems that have non-intrusive fixes,' in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, IEEE, vol. 1, 2015, pp. 902–912 (cit. on p. 107).

[130] J. P. Sandoval Alcocer, A. Bergel and M. T. Valente, 'Learning from source code history to identify performance failures,' in *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, 2016, pp. 37–48 (cit. on pp. 107–109).

[131] L. Jiang, Z. Su and E. Chiu, 'Context-based detection of clone-related bugs,' in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2007, pp. 55–64 (cit. on pp. 107, 108).

[132] R. D. Banker, S. M. Datar and D. Zweig, 'Software complexity and maintainability,' *Age*, vol. 11, no. 5.6, p. 3, 1989 (cit. on p. 107).

[133] T. Aslam, 'A taxonomy of security faults in the unix operating system,' *Master's thesis, Purdue University*, vol. 199, no. 5, 1995 (cit. on p. 108).

[134] Y Levendel, 'Defects and reliability analysis of large software systems: Field experience,' in *1989 The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*, IEEE Computer Society, 1989, pp. 238–239 (cit. on p. 108).

[135] E. Razina and D. S. Janzen, 'Effects of dependency injection on maintainability,' in *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications: Cambridge, MA*, 2007, p. 7 (cit. on p. 108).

[136] A. Sawant, P. H. Bari and P. Chawan, 'Software testing techniques and strategies,' 2012 (cit. on pp. 108, 109).

[137] Z. Yan, D. Guowei, G. Tao and Y. Jianyu, 'Taxonomy of source code security defects based on three-dimension-tree,' in *International Conference on Computer and Computing Technologies in Agriculture*, Springer, 2013, pp. 232–241 (cit. on p. 109).

[138] M. Felderer, B. Marculescu, F. G. de Oliveira Neto, R. Feldt and R. Torkar, 'A testability analysis framework for non-functional properties,' in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, IEEE, 2018, pp. 54–58 (cit. on p. 116).

[139] P. Morrison, K. Herzig, B. Murphy and L. Williams, 'Challenges with applying vulnerability prediction models,' in *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*, 2015, pp. 1–9 (cit. on p. 116).

[140] K. Al-Sabbagh, M. Staron, R. Hebig and F. Gomes, 'A classification of code changes and test types dependencies for improving machine learning based test selection,' in *Proceedings of the 17th Int. Conference on Predictive Models and Data Analytics in Software Engineering*, 2021, pp. 40–49 (cit. on pp. 121, 123).

[141] O. Dahiya and K. Solanki, 'A systematic literature study of regression test case prioritization approaches,' *Int. Journal of Engineering & Technology*, vol. 7, no. 4, pp. 2184–2191, 2018 (cit. on p. 122).

[142] J. Chi, Y. Qu, Q. Zheng *et al.*, 'Relation-based test case prioritization for regression testing,' *Journal of Systems and Software*, vol. 163, p. 110 539, 2020 (cit. on p. 122).

[143] F. G. de Oliveira Neto, A. Ahmad, O. Leifler, K. Sandahl and E. Enoiu, 'Improving continuous integration with similarity-based test case selection,' in *Proceedings of the 13th International Workshop on Automation of Software Test*, 2018, pp. 39–45 (cit. on p. 122).

[144] A. Bertolino, A. Guerriero, B. Miranda, R. Pietrantuono and S. Russo, 'Learning-to-rank vs ranking-to-learn: Strategies for regression testing in continuous integration,' in *Proceedings of the ACM/IEEE 42nd Int. Conference on Software Engineering*, 2020, pp. 1–12 (cit. on p. 122).

[145] A. Orso, N. Shi and M. J. Harrold, 'Scaling regression testing to large software systems,' *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 6, pp. 241–251, 2004 (cit. on p. 123).

[146] G. E. Batista, R. C. Prati and M. C. Monard, 'A study of the behavior of several methods for balancing machine learning training data,' *ACM SIGKDD explorations newsletter*, vol. 6, no. 1, pp. 20–29, 2004 (cit. on pp. 125, 146).

[147] P. S. Bayerl and K. I. Paul, 'What determines inter-coder agreement in manual annotations? a meta-analytic investigation,' *Computational Linguistics*, vol. 37, no. 4, pp. 699–725, 2011 (cit. on p. 128).

[148] A Esuli and F Sebastiani, 'Proceedings of the 5th conference on language resources and evaluation,' 2006 (cit. on p. 128).

[149] M. Zolfagharinia, B. Adams and Y.-G. Guéhénuc, 'Do not trust build results at face value-an empirical study of 30 million cpan builds,' in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, IEEE, 2017, pp. 312–322 (cit. on p. 137).

[150] M. Beller, G. Gousios and A. Zaidman, 'Oops, my tests broke the build: An analysis of travis ci builds with github,' PeerJ Preprints, Tech. Rep., 2016 (cit. on pp. 137, 139).

[151] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian and R. Bowdidge, 'Programmers' build errors: A case study (at google),' in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 724–734 (cit. on p. 137).

[152] F. Hassan and X. Wang, 'Change-aware build prediction model for stall avoidance in continuous integration,' in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, IEEE, 2017, pp. 157–162 (cit. on pp. 137, 139, 140).

[153] A. Ni and M. Li, 'Cost-effective build outcome prediction using cascaded classifiers,' in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, IEEE, 2017, pp. 455–458 (cit. on pp. 139, 140).

[154] J. Xia, Y. Li and C. Wang, 'An empirical study on the cross-project predictability of continuous integration outcomes,' in *2017 14th Web Information Systems and Applications Conference (WISA)*, IEEE, 2017, pp. 234–239 (cit. on p. 139).

[155] T. Rausch, W. Hummer, P. Leitner and S. Schulte, 'An empirical analysis of build failures in the continuous integration workflows of java-based open-source software,' in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, IEEE, 2017, pp. 345–355 (cit. on p. 139).

[156] Y. Luo, Y. Zhao, W. Ma and L. Chen, 'What are the factors impacting build breakage?' In *2017 14th Web Information Systems and Applications Conference (WISA)*, IEEE, 2017, pp. 139–142 (cit. on p. 139).

[157] J. Yao and M. Shepperd, 'Assessing software defection prediction performance: Why using the matthews correlation coefficient matters,' in *Proceedings of the Evaluation and Assessment in Software Engineering*, 2020, pp. 120–129 (cit. on p. 144).

[158] M. Feurer and F. Hutter, 'Hyperparameter optimization,' in *Automated machine learning*, Springer, Cham, 2019, pp. 3–33 (cit. on p. 145).

[159] M. Kuutila, M. Mäntylä, U. Farooq and M. Claes, 'Time pressure in software engineering: A systematic review,' *Information and Software Technology*, vol. 121, p. 106 257, 2020 (cit. on p. 155).

[160] N. Pritam, M. Khari, R. Kumar *et al.*, 'Assessment of code smell for predicting class change proneness using machine learning,' *IEEE Access*, vol. 7, pp. 37 414–37 425, 2019 (cit. on p. 155).

[161] A. Hovsepyan, R. Scandariato, W. Joosen and J. Walden, 'Software vulnerability prediction using text analysis techniques,' in *Proceedings of the 4th international workshop on Security measurements and metrics*, 2012, pp. 7–10 (cit. on p. 155).

[162] K. Al-Sabbagh, M. Staron and R. Hebig, 'Predicting build outcomes in continuous integration using textual analysis of source code commits,' in *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2022, pp. 42–51 (cit. on pp. 155, 166).

[163] J. Van Hulse, T. M. Khoshgoftaar, C. Seiffert and L. Zhao, 'Noise correction using bayesian multiple imputation,' in *2006 IEEE International Conference on Information Reuse & Integration*, IEEE, 2006, pp. 478–483 (cit. on p. 156).

[164] T. M. Khoshgoftaar, N. Seliya and K. Gao, 'Detecting noisy instances with the rule-based classification model,' *Intelligent Data Analysis*, vol. 9, no. 4, pp. 347–364, 2005 (cit. on p. 156).

[165] D. R. Wilson and T. R. Martinez, 'Reduction techniques for instance-based learning algorithms,' *Machine learning*, vol. 38, no. 3, pp. 257–286, 2000 (cit. on p. 158).

[166] S. Kim, H. Zhang, R. Wu and L. Gong, 'Dealing with noise in defect prediction,' in *2011 33rd International Conference on Software Engineering (ICSE)*, IEEE, 2011, pp. 481–490 (cit. on p. 159).

[167] G. Liebchen, B. Twala, M. Shepperd, M. Cartwright and M. Stephens, 'Filtering, robust filtering, polishing: Techniques for addressing quality in software data,' in *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, IEEE, 2007, pp. 99–106 (cit. on pp. 159–161).

[168] S. Zhong, T. M. Khoshgoftaar and N. Seliya, 'Analyzing software measurement data with clustering techniques,' *IEEE Intelligent Systems*, vol. 19, no. 2, pp. 20–27, 2004 (cit. on pp. 159, 160).

[169] C. Seiffert, T. M. Khoshgoftaar, J. Van Hulse and A. Folleco, 'An empirical study of the classification performance of learners on imbalanced and noisy software quality data,' *Information Sciences*, vol. 259, pp. 571–595, 2014 (cit. on pp. 159, 160).

[170] C. Sadowski, E. Söderberg, L. Church, M. Sipko and A. Bacchelli, 'Modern code review: A case study at google,' in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, 2018, pp. 181–190 (cit. on p. 161).

[171] D. Chicco and G. Jurman, 'The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation,' *BMC genomics*, vol. 21, no. 1, pp. 1–13, 2020 (cit. on p. 166).

[172] F. Huq, M. Hasan, M. A. P. Haque, S. Mahbub, A. Iqbal and T. Ahmed, *Review4Repair: Code Review Aided Automatic Program Repairing*, version 1.0, Jan. 2021. DOI: `10.5281/zenodo.4445747`. [Online]. Available: `https://doi.org/10.5281/zenodo.4445747` (cit. on p. 167).

[173] K. W. Al-Sabbagh, M. Staron and R. Hebig, 'Improving test case selection by handling class and attribute noise,' *Journal of Systems and Software*, vol. 183, p. 111 093, 2022 (cit. on pp. 168, 191).

[174] C Bentéjac, A Csörgo and G Martínez-Muñoz, 'A comparative analysis of xgboost,' *ArXiv abs*, 1911 (cit. on p. 171).

[175] Y. Singh, P. K. Bhatia, A. Kaur and O. Sangwan, 'Application of neural networks in software engineering: A review,' in *Information Systems, Technology and Management: Third International Conference, ICISTM 2009, Ghaziabad, India, March 12-13, 2009. Proceedings 3*, Springer, 2009, pp. 128–137 (cit. on p. 171).

[176] N. E. Fenton and M. Neil, 'A critique of software defect prediction models,' *IEEE Transactions on software engineering*, vol. 25, no. 5, pp. 675–689, 1999 (cit. on p. 186).

[177] J. Mendoza, J. Mycroft, L. Milbury, N. Kahani and J. Jaskolka, 'On the effectiveness of data balancing techniques in the context of ml-based test case prioritization,' in *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2022, pp. 72–81 (cit. on p. 190).