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.

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

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 [1].

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 [2] 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 [3] 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 [4]. Two categories of noise commonly discussed in the literature are class and attribute noise [5]. Class noise arises from contradictory or mislabelled instances in the training data, while attribute noise occurs when attributes contain irrelevant or missing information [6].

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 [5]. 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 [7]. 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.

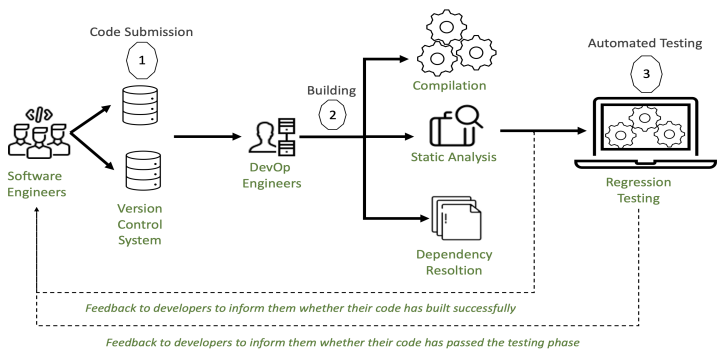

Figure 1.1: The continuous integration process.

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 [8].

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 [9]. 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 [9]. 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 [10].

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

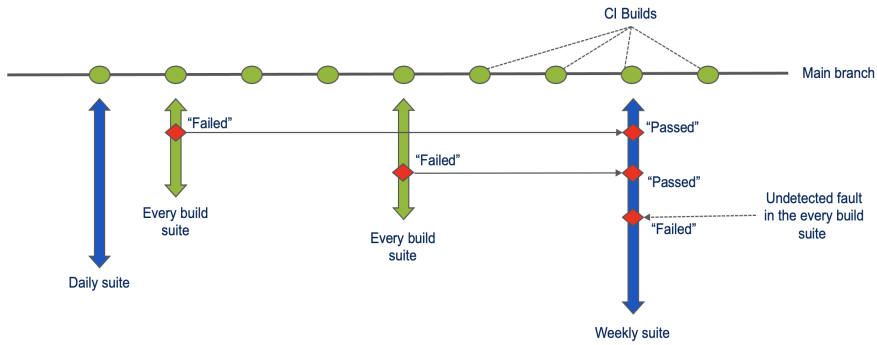

Figure 1.2: An illustration of regression testing in CI.

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:

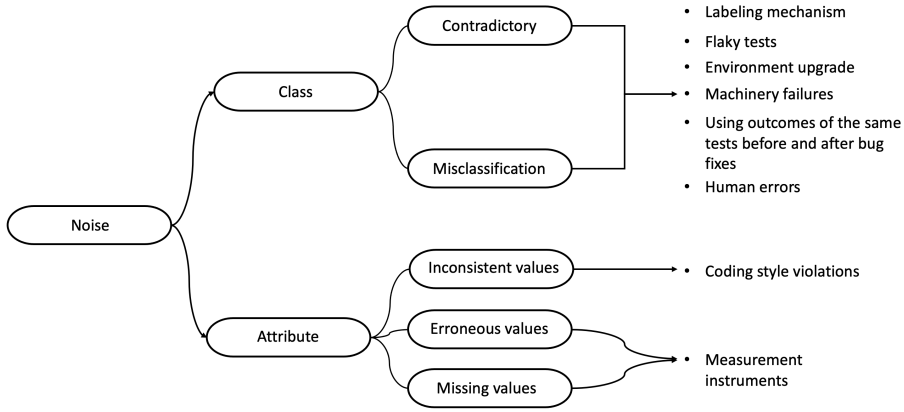

Figure 1.3: An overview of noise types and their sources in CI.

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 [11]. 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. [12], 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 [13], [14], [15], and [16].

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 [13], 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 [12]. 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 [17].

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 [18]. 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 [12]).

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 [19] and [14], 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 [12].

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 [20], [21], and [22], 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 [23]. Based on the majority voting mechanism presented in [20], 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 [24] and [25]. 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 [26].

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 [27] (e.g., weighted methods per class, number of ancestors of a class), process metrics [28] (e.g., code churns, number of changes made to files in commits), product metrics [29] (e.g., total size of the program, number of lines of code in a commit), and structural metrics [30] (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% [31] and [32]. In the context of CI, this argument seems attainable by several ML-based approaches. For example, Saidani et al. [7] 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. [33] 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. [34] 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

[29]. 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 [35] 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. [36] 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 [37] 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. [14] 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., [36] and [38]).

Guan et al. [13] 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. [39] 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. [40] 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. [6], 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 [41], 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 [12] 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 **??**.

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 [42]. 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:

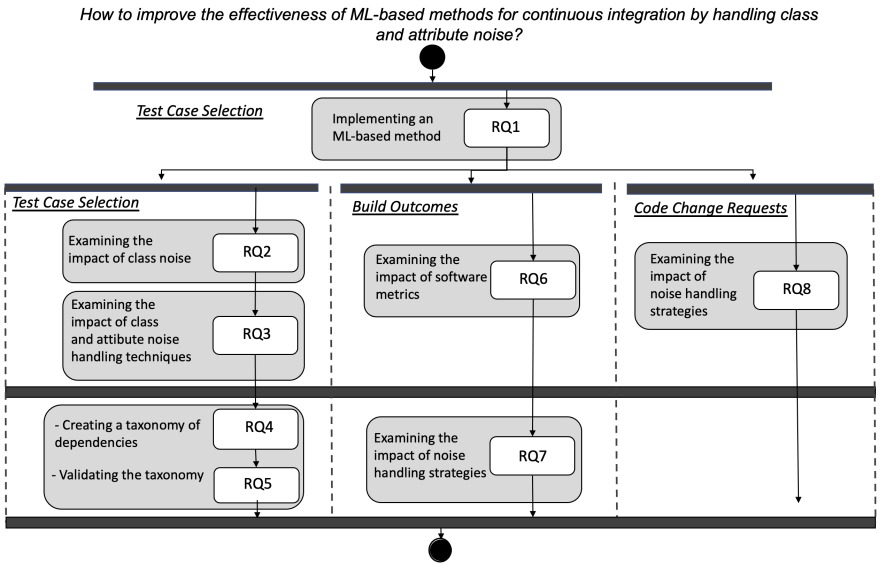

Figure 1.4: Mapping between research questions.

- *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 [43]. 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. [44] 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 [45]. 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. [45] 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

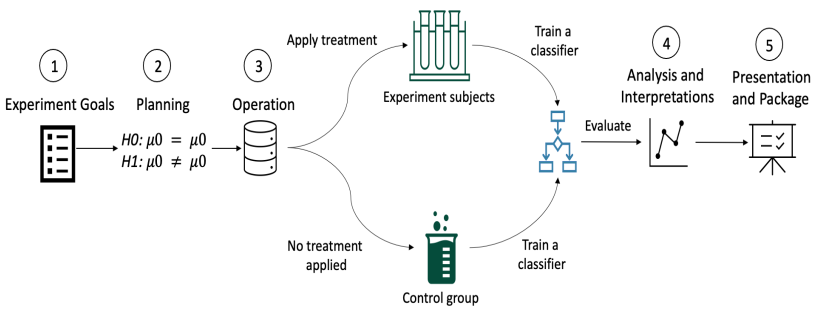

Figure 1.5: An overview of the controlled experiment design used in the thesis.

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 [46]. 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 [12]. 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 [47]. 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 [48]. 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. [45] 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 [49][50], 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. [51] 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 [52] 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 [20][53], 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 [21]. Nevertheless, experimental observations also showed that having more diversity in the ensemble of classifiers in CF leads to achieving higher precision [51]. 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 [54] 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.

Bibliography

[1] 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 p. 1).

[2] 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 p. 1).

[3] 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 p. 2).

[4] 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).

[5] 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 p. 2).

[6] 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).

[7] 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).

[8] 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).

[9] 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).

[10] 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).

[11] 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).

[12] 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).

[13] 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).

[14] 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).

[15] 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 p. 9).

[16] 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 p. 9).

[17] 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 p. 9).

[18] 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).

[19] C.-M. Teng, 'Correcting noisy data.,' in *ICML*, Citeseer, 1999, pp. 239–248 (cit. on p. 9).

[20] 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).

[21] 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).

[22] 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).

[23] 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 p. 11).

[24] 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).

[25] 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).

[26] 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).

[27] 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).

[28] 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).

[29] 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).

[30] 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).

[31] 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).

[32] 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).

[33] 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).

[34] 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).

[35] 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).

[36] 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).

[37] 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).

[38] 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 p. 13).

[39] 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 p. 13).

[40] 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 p. 14).

[41] 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 p. 15).

[42] 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).

[43] 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).

[44] 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).

[45] 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).

[46] 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).

[47] 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 p. 24).

[48] 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 p. 25).

[49] 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).

[50] 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).

[51] 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] Y. Ma and H. He, 'Imbalanced learning: Foundations, algorithms, and applications,' 2013 (cit. on p. 33).

[53] 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 p. 33).

[54] 'Iso/iec/ieee international standard - software and systems engineering –software testing–part 1: Concepts and definitions,' Tech. Rep., 2020, pp. 1–50 (cit. on p. 35).