Thesis for The Degree of Licentiate of Engineering

Noise Handling For Improving Machine Learning Based Test Case Selection

Khaled Walid Al-Sabbagh

Department of Computer Science & Engineering
Division of Interaction Design and Software Engineering
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden, 2021

Noise Handling For Improving Machine Learning Based Test Case Selection

Khaled Walid Al-Sabbagh

Department of Computer Science & Engineering
Division of Interaction Design and Software Engineering
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden

This thesis has been prepared using LaTeX.
Printed by Chalmers Reproservice,
Gothenburg, Sweden 2021.

"Live as if you were to die tomorrow. Learn as if you were to live forever."
- Mahatma Gandhi.

Abstract

Background: Continuous integration is a modern software engineering practice that promotes rapid integration and testing of code changes as soon as they get committed to the project repository. One challenge in adopting this practice lies in the long time required for executing all available test cases to perform regression testing. The availability of large amounts of data about code changes and executed test cases in continuous integration systems poses an opportunity to design data-driven approaches that can effectively select a subset of test cases for regression testing.

Objective: The objective of this thesis is to create a method for selecting test cases that have the highest probability of revealing faults in the system, given new code changes pushed into the code-base. Using historically committed source code and their respective executed test cases, we can utilize textual analysis and machine learning to design a method, called MeBoTs, that can learn the selection of test cases.

Method: To address this objective, we carried out two design science research cycles and two controlled experiments. A combination of quantitative and qualitative data collection methods were used, including testing and code commits data, surveys, and a workshop, to evaluate and improve the effectiveness of MeBoTs in selecting effective test cases.

Results: The main findings of this thesis are that: 1) using an elimination and a relabelling strategy for handling class noise in the data increases the performance of MeBoTs from 25% to 84% (F1-score), 2) eliminating attribute noise from the training data does not improve the predictive performance of a test selection model (F1-score remains unchanged at 66%), and 3) memory management changes in the source code should be tested with performance, load, soak, stress, volume, and capacity tests; the algorithmic complexity changes should be tested with the same tests for memory management code changes in addition to maintainability tests.

Conclusion: Our first conclusion is that textual analysis of source code can be effective in test case selection if a class noise handling strategy is applied for curating incorrectly labeled data points in the training data. Secondly, test orchestrators do not need to handle attribute noise in the data, since it does not lead to an improvement in the performance of MeBoTs. Finally, we conclude that the performance of MeBoTs can be improved by instrumenting a tool that automatically associates code changes of specific types to test cases that are in dependency for training.

Keywords

Test Case Selection, Continuous Integration, Machine Learning, Textual Analysis

Acknowledgment

Through the support of many individuals, I have managed to be here today, realizing a major milestone in my academic career. First and foremost, I would like to thank my supervisors, Miroslaw Staron and Regina Hebig, for the mentorship and support that they have given me. I deeply appreciate their patience and guidance. I will remain forever thankful to my fountain of cherish and support Joumana, without whom this thesis could not be finished. To my astonishingly strong family in Syria and the United States; my mother, sisters, uncle, and brother who have been my rock from different sides of the planet. I am also thankful to my second family in Sweden, the Bradley's, who made me perceive Sweden as my first homeland. My acknowledgement and gratitude also go to a long list of friends and colleagues, including Khaled Khaled, for showing me how to cherish life, no matter how imperfect it gets; to Francisco Gomez for always being available to support and provide valuable advice; to Alaa Al-Nuweiri for the long walks and deep conversations, to Yasmine Moussalli, Rasha Al-Sabbagh for the positive vibes they disseminated during my short visits to Syria. Last but not the least, I would like to thank all my colleagues in the division of Interaction Design and Software Engineering and in the metric team at Software Center.

List of Publications

Appended publications

This thesis is based on the following publications:

[A] KW. Al-Sabbagh, M. Staron, R. Hebig, W. Meding "Predicting Test Case Verdicts Using Textual Analysis of Committed Code Churns" *IWSM-Mensura. 2019, pp. 138–153.*

[B] KW. Al-Sabbagh, M. Staron, R. Hebig "The Effect of Class Noise on Continuous Test Case Selection: A Controlled Experiment on Industrial Data" *International Conference on Product-Focused Software Process Improvement. Springer. 2020, pp. 287–303.*

[C] KW. Al-Sabbagh, M. Staron, R. Hebig "Improving test case selection by handling class and attribute noise" *Accepted for publication in Journal of Systems and Software.*

[D] KW. Al-Sabbagh, M. Staron, R. Hebig, F Gomez "A classification of code changes and test types dependencies for improving machine learning based test selection" *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering. PROMISE 2021. Athens, Greece: Association for Computing Machinery, 2021, pp. 40–49.*

Other publications

The following publications were published before and during my Ph.D studies, but were not included in this thesis due to contents overlapping that of appended publications or contents not related to the thesis.

[a] 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.

[b] 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).

[c] 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.

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

[e] 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.

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

Research Contribution

We adopted the CRediT (Contribution Roles Taxonomy) model, proposed by Brand et al. [1], to define the authors' contributions to the appended publications in this thesis. Table 1 summarizes these contributions.

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

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

Contents

Chapter 1

Introduction

As software is becoming more pervasive in everyday life, contemporary software companies need to keep up with the growing pace of market demands to deliver complex features at higher quality and lower cost. In an attempt to accommodate for these growing demands, companies have started to increasingly adopt the practice of continuous integration (CI), which advocates for continuously building and testing newly pushed code changes at frequent time intervals [2]. A well-known challenge that impedes the success of CI concerns the testing activities required at each integration cycle [3]. One example of a testing activity is regression testing, which seeks to verify that no new faults in existing software artifacts arise as a consequence of introducing new code changes. This type of testing is central for assuring software quality and reducing cost overruns in software development projects. A straightforward approach for leveraging confidence about the soundness of code changes is to execute all existing set of test cases in a suite [4]. However, the consequent effect of using this approach is a prolonged feedback loop cycle between the CI system and software developers, and an increased need for hardware resources to test code submissions made at frequent time intervals.

To address the problem of cost overruns in regression testing, several test case selection (TCS) approaches have been proposed [5], [6], [7], [8]. These approaches seek to reduce the size of regression suites by finding an effective subset of test cases from the pool of reusable tests. Despite the recent advancements in the area of TCS, the majority of existing techniques advocate for selecting test cases based on test coverage criteria, which are reported to produce limited success, as they involve costly static code analysis [9], [10], [11], [12], [13].

In the context of CI, the number of executed test cases gets large as new code changes get integrated into the development branch several times every day. For example, a recent study reports that in Google, more than 20 code changes are submitted by developers every minute, which leads to executing 1.5 million test cases every day [14]. This large volume of data available from CI systems poses an opportunity to design data-driven approaches that investigate the relation between pushed code changes and test execution outcomes (Passed/Failed). One strategy for utilizing the CI data is the use of machine learning (ML) for predicting which subset of test cases should be executed given newly submitted code changes.

However, applying ML for TCS is hindered by the quality of the underlying data, which often comes with irrelevant or meaningless data known as noise [15]. The effects that noise has on ML models range from a prolonged time of learning to a lower classification accuracy [16]. Two categories of noise are reported in the body of literature: class and attribute noise. Class noise (also known as label or annotation noise) occurs as a result of either having contradictory or mislabelled data-points in the training data, whereas attribute noise occurs when the attributes contain irrelevant or missing information [15]. In the context of TCS, class noise can be observed when, for example, a code line in the data appears more than once with different class labels (test outcomes) for the same test. These duplicate appearances for the same line become class noise for ML models that might potentially hamper their classification performance. Similarly, attribute noise can be observed, when lines of code in the system under test appear to deviate in their attributes from the majority of similar lines. Such deviations in the attributes of lines can be become noise to the ML model if, for example, they indicate spurious correlations that coincidentally appear in the training data only, and not as often in unseen lines of code. Accordingly, handling noise in lines that belong to the less representative set of data points is here hypothesized to reduce the effect of learning spurious correlations, and ultimately improve the learning performance of ML models for TCS. To address the problem of noise, both in the class and attributes, several research studies proposed noise handling strategies that can be classified under three broadly categories: 1) tolerance, 2) elimination, and 3) correction [17], [18], [19], [20]. In this research, the term "noise handling" refers to the reduction in the number of noisy lines of code that appear in the training data, which is here achieved by using a combination of eliminating and correction based approaches. The high-level goal of this thesis is to create a method for selecting test cases that have the highest probability of revealing faults in the system under test, given new code changes pushed into the code-base. The general research question that this thesis aims to answer is the following:

How to improve the performance of ML-based test case selection models by handling noise?

This thesis 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. After that, Section 1.2 presents related studies that shed light on existing test case selection and noise handling techniques. In Section 1.3, the research design of the included papers is described. Section 1.4 presents the main findings that were drawn from each of the included papers. Section 1.5 answers the general research question. Section 1.6 discusses the threats to the validity. Sections 1.7 and 1.8 conclude the results and provide an outlook for future work.

1.1 Theoretical Framework

This section introduces key concepts and code examples that are necessary to allow the reader of this thesis to understand its content. We begin by describing

the use of test case selection techniques. After that, we demonstrate, through a code example, the dependency between test execution outcome and code changes. Then, we define the existing types of noise that were examined for an effect on test case selection in this research. Finally, we present and define the set of code change categories and test case types that were investigated for a dependency in this research.

1.1.1 Test Case Selection

Executing large test suites is undesirable in software development projects since it requires both time and computer resources [21]. Therefore, it is important to avoid executing unnecessary tests (passing tests) and strive to execute tests that have high potentials in revealing faults (failing tests). Test case selection is an approach that aims at addressing this problem of large test suite by searching for a subset of effective test cases according to a criterion of interest. Minimizing the size of test suite becomes more essential in CI, as it allows developers to quickly fix faults as early as new code changes are pushed into the development repository. Figure 1.1 illustrates a timeline for executing several test suites that are set up to continuously test every integration made by software developers. The CI system tries to reduce the size of the executed suit after every build/commit, represented by circles, using a test case selection approach.

According to Narciso et al. [21], test case selection covers two main paradigms. In the first paradigm, the criteria of test selection is based on the modifications made to the program under test, where the selection of tests depends on their relevance to the modified code. The second paradigm uses heuristics, such as similarity based criteria between test cases and code coverage to select test cases. The hypothesis on which heuristic approaches build on is that the more diverse the selected test cases, the higher possible it becomes to detect faults in the source code. Our research belongs to the first paradigm of approaches for test selection, where tests are selected based on their relevance to the modified parts in the code.

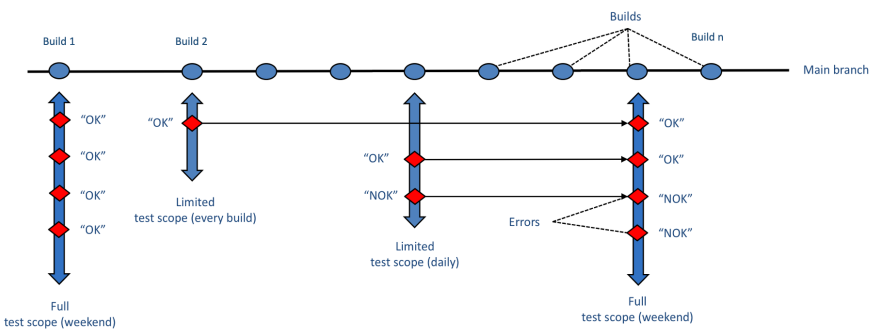


Figure 1.1: An illustration of test case selection in continuous integration.

1.1.2 Example of Dependency Between Test Case Outcome and Code Changes

This research builds on the assumption that test case execution outcomes are dependent on changes made in the code base. In this section, a C++ code example is provided to illustrate what we mean by a dependency. In this research, a dependency is defined as a reaction of one or more test case types (e.g., performance and security) to a specific set of changes made in code revisions. A revision is a unique identifier that gets created by a version control system(e.g., git) to identify a set of code changes made by developers at a particular time. The set of changes referenced by each revision is then retrieved and compiled by the CI server (e.g., Jenkins) to create a "build". Therefore, a build comprises the set of code changes that were made in a revision.

Figure 1.2 shows two code fragments that belong to a C++ program. The `getNetSalary` function in revision 1 instantiates a pointer to the class `Leave` using the `new` code construct. The function then computes the net salary of a given employee `empId` based on the number of leaves that was registered for the designated employee. The code fragment in revision 2 is a modified version of the program, containing a set of changes made to function `getNetSalary`. The framed lines in revision 2 are the changed lines.

`performanceTestGetSalary` is a performance test case that verifies whether the function `getNetSalary` can retrieve the net salary of 10,000 employees in less than 1,500 micro seconds. To do this, `performanceTestGetSalary` measures the elapsed time between the beginning and end of the function call for 10,000 employees. Assuming that the test case `performanceTestGetSalary` was one of several test cases that got executed in the regression suites of builds 1 and 2 in Figure 1.1, then the following test outcomes will be recorded <*Revision$_1$: failed, Revision$_2$: passed*>. These test outcomes are not surprising, since the code in revision 1 creates the object `Leave` by allocating memory in a heap-space that never gets freed from memory (e.g., using the `delete` keyword). In contrast, the memory allocation in revision 2 was modified to become stack-based, which is advantageous over heap-based since the allocation is done by the compiler. In other words, the size of memory to be allocated for object `Leave` is known to the compiler and gets automatically de-allocated after the function call is over. Therefore, the reaction of test case `performanceTestGetSalary` (i.e., from failed to passed) in revision 2 can be explained by the change in memory allocation from heap-based to stack-based. We use the term dependency to refer to such reactions of test cases to new changes in the code base.

1.1.3 Class and Attribute Noise in Code Changes

Data are considered as the most important input for empirical software engineering research, since they aid the discovery of new strategies and support practitioners to make strategic decisions [15]. Existing reports suggest that 60 to 95% of the effort on data analysis is spent on data cleaning activities [22]. Among the most common activities is to handle noise in the data, which, in this research, is defined as data points that come with irrelevant or meaningless values. The inaccuracies can be found either in the attributes (independent variables), the class labels (dependent variable), or in both. Accordingly, noise

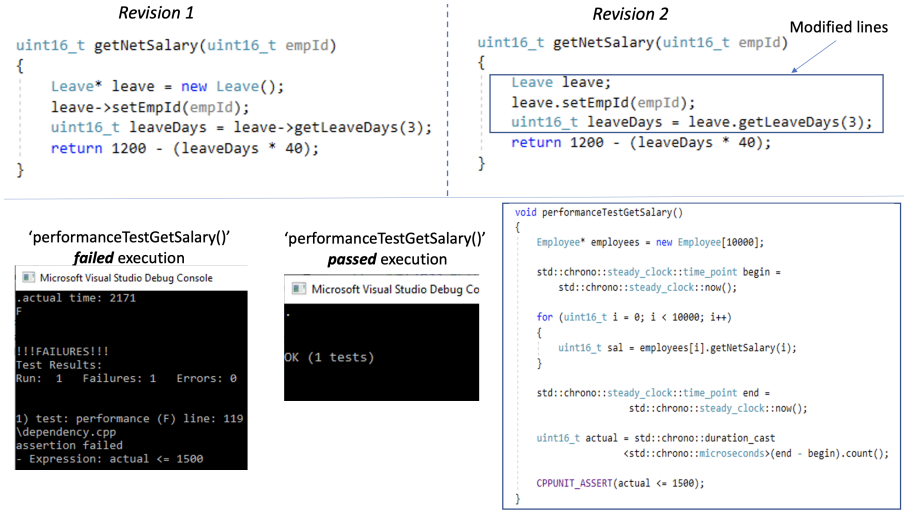


Figure 1.2: An example illustrating the dependency between a performance test case and a change in the code base

can be categorized under two categories; attribute and class noise.

The effect of both categories of noise has been intensively studied in different domain areas [23], [16], [24], and a general consensus about their negative impact on ML models has been reported, as pointed out in [15]. Figure 1.3 provides an overview of causes that trigger both categories of noise.

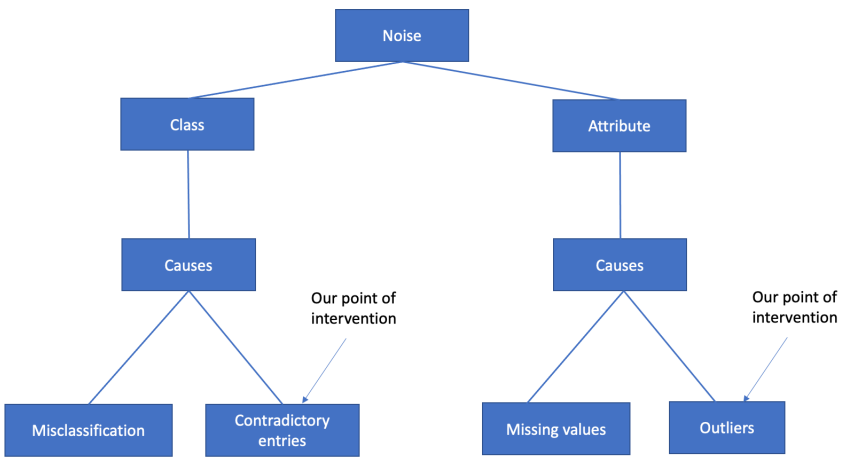


Figure 1.3: An overview of noise categories and their causes.

In this research, we study the impact of class and attribute noise on the performance of the ML model for TCS with respect to two causes - contradictory entries and outliers. Contradictory entries are one or more duplicate lines of code that appear with different test execution results (class values), whereas

outliers are exceptional values in the attributes of lines of code in comparison with the rest of similar lines. The purpose is to gain an understanding of whether handling noise can improve the predictive performance of the TCS model.

1.1.4 Class Noise in Test Case Selection

To illustrate the problem of class noise in the context of test case selection, Figures 4.3 exemplifies a scenario in which class noise might occur in a software program. The equivalent vector representations of the majority of lines in the program and their class labels are illustrated in the matrix to the right side of Figure 4.3. Details about how these line vectors are derived can be found in Papers A and B. The class label of each line vector in the Figure is determined by the outcome of a test case that was executed against the program revisions in CI. In the example presented in Figure 4.3, lines 1 to 12 belong to the same program revision, and their class label (‘0’) was determined by the execution result of a test case in which the revision failed during CI. Similarly, lines 13 to 17 belong to another program revision, and their class label (‘1’) was determined by a test case in which the revision passed during CI. These line vectors and their class labels are then fed into a machine ML model to predict whether lines of code in future program revisions will trigger a test case failure or pass.

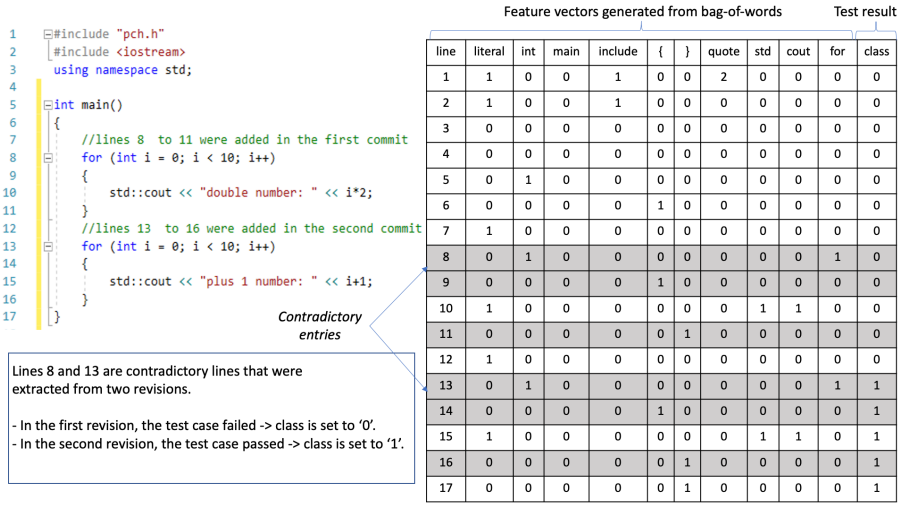


Figure 1.4: An example of class noise with two contradictory data points.

The shaded lines in the matrix of feature vectors (lines 8 and 13), (lines 9 and 14), and (lines 11 and 16) are three pairs of duplicate line vectors, where each line in a pair is assigned a different class label than its duplicate line in the pair. Following this description and illustration of class noise, the formula for calculating the ratio of class is:

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

Since the example program in Figure 4.3 has a total of six contradictory lines, the formula for calculating the noise ratio for this example is thus:

$$\text{Noise ratio} = \frac{6}{17} = 0.35$$

1.1.5 Noise Handling Strategies

Approaches for automatically handling noise in the data can be categorized under three broad categories, as pointed out in [17], [18], [19], [20]. These categories are as follows:

[a] Tolerance: In this category of approaches, noisy data points are dealt with by leaving the noise in place and designing ML algorithms that can tolerate a maximum threshold of noise.

[b] Elimination: Approaches in this category identify noisy data points that are suspected for being mislabelled or redundant, and remove them from the training data.

[c] Correction: In the category of approaches, instead of removing the undesired data points, these points get corrected by replacing their values with more appropriate ones.

Table 1.1 summarizes the advantages and disadvantages associated with each category of approaches. In the tolerance category, the negative impact of noisy data points is handled by designing robust algorithms that are assumed to be less sensitive to noise than others - typically using tree pruning and rule truncation [17]. For example, the C4.5 algorithm uses pruning strategies to eliminate statistically insignificant parts of the tree to construct the final model [25]. The most prominent advantage of this category is the fact that no time needs to be invested in cleaning the data. Conversely, relying on the robustness of algorithms to handle noisy points has been reported to have a limited success when the noise level exceeds a threshold, as pointed out in [26]. That is, even robust algorithms may encounter poor learning performance if the noise level gets relatively high.

Compared with other noise handling categories, elimination based approaches are computationally expensive due to their iterative nature, where in every iteration only one or a few points can be detected as noisy [27]. In addition, elimination based approaches have a tendency of removing data points that are not actually noisy. For example, a data point that comes with an incorrect class label and is in contradiction with one or more duplicate data points might be detected and removed. This means that we compromise information loss at the interest of retaining a data set with non-contradictory data points. On the other hand, an advantage of this category over the others is the explicit detection of potentially noisy points and the plausibility of showing them to the user, who can decide whether the detected points should be removed from the data or not [27].

In the last category of approaches (i.e., correction), noisy data points are corrected rather than removed or left in place. Thus, much of the information from the originally collected data-set would be preserved. The major drawback

of correcting noisy data points is the relatively high time complexity involved in existing approaches, such as [28], [18]. Another disadvantage is the fact that we risk introducing bias towards one of the classes when correcting the class labels of noisy points. Finally, existing correcting approaches operate in a supervised ML environment, disqualifying their use when the class label is unavailable [25].

Table 1.1: Advantages and disadvantages of existing noise handling approaches

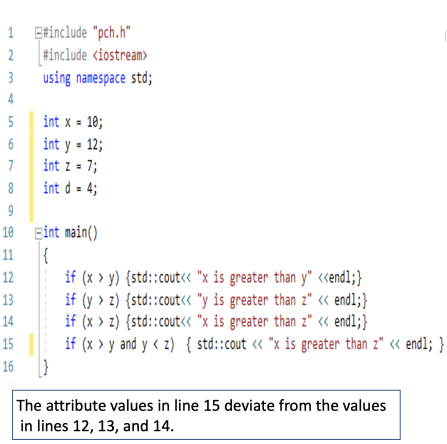
	Tolerance	Elimination	Correction
Pros	- No time is needed to handle noisy data. - No information loss.	- Explicit detection of noisy data points.	- No information loss.
Cons	- Reduces the performance of classifiers as the noise ratio increases.	- High computational cost to detect and remove noisy data.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

1.1.6 Attribute Noise in Test Case Selection

In addition to studying the impact of class noise, we also studied the impact of attribute noise on the performance of the ML model in MeBoTS.

Figure 4.4 is a C++ example program that illustrates a scenario in which attribute noise is incurred. Lines 12, 13, 14, are three `if` statements, each containing one conditional expression that guards the execution of their scopes. Line 15 is a fourth `if` statement in the same code fragment, containing two conditional expressions. By examining the corresponding matrix of feature vectors in the figure, we notice that the attribute values of line 15 deviate substantially from the majority of `if` statements in the program. We refer to such lines that deviate from the majority of similar lines in the code as noisy with respect to the attributes, since they seldom appear in the data. To identify lines of code that come with high attribute noise, we used an existing algorithm from the literature, called PANDA. The PANDA algorithm identifies such data points by comparing pairs of attributes in the space of feature vectors. The output is an ordered list of noise scores for each line of code - the higher the noise score for a line, the higher it deviates from normal. Upon ranking noisy instances, the generated list can be used to eliminate lines of code that come with the highest rank with respect to attribute noise.

The algorithm starts by iterating through all attributes in the input feature vectors. During each iteration, a single attribute x_j gets partitioned into a number of bins, set by the user. Each bin will have the same amount of data



```
1  #include "pch.h"
2  #include <iostream>
3  using namespace std;
4
5  int x = 10;
6  int y = 12;
7  int z = 7;
8  int d = 4;
9
10 int main()
11 {
12    if (x > y) {std::cout<< "x is greater than y" <<endl;}
13    if (y > z) {std::cout<< "y is greater than z" << endl;}
14    if (x > z) {std::cout<< "x is greater than z" << endl;}
15    if (x > y and y < z)  { std::cout << "x is greater than z" << endl; }
16 }
```

The attribute values in line 15 deviate from the values in lines 12, 13, and 14.

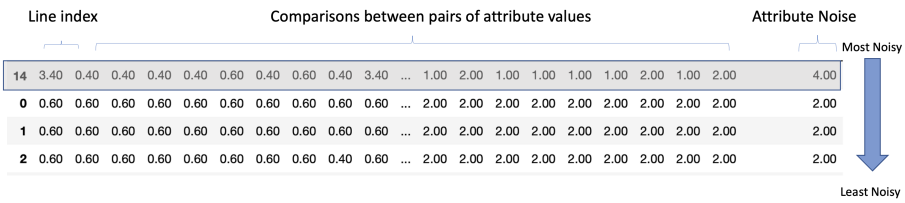
Feature vectors generated from bag-of-words

Test result

line	if	var	(	)	{	}	<	>	and	;	class
1	0	0	0	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	0	0	0	0	1
3	0	0	0	0	0	0	0	0	0	1	1
4	0	0	0	0	0	0	0	0	0	0	1
5	0	1	0	0	0	0	0	0	0	1	1
6	0	1	0	0	0	0	0	0	0	1	1
7	0	1	0	0	0	0	0	0	0	1	1
8	0	1	0	0	0	0	0	0	0	1	1
9	0	0	0	0	0	0	0	0	0	1	1
10	0	1	1	1	0	0	0	0	0	1	1
11	0	0	0	0	1	0	0	0	0	1	1
12	1	2	1	1	1	1	0	1	0	1	1
13	1	2	1	1	1	1	0	1	0	1	1
14	1	2	1	1	1	1	0	1	0	1	1
15	1	4	1	1	1	1	1	1	1	1	1

Figure 1.5: An example of attribute noise with one outlier line of code.

points, given that the number of input points is divisible by the number of partitions. In the absence of tied values, the algorithm includes all boundary instances that fall outside the range of the bin size in the last bin. After the partitioning is complete, the mean and standard deviation for instances in each bin are calculated and used to derive a standardized value for each instance in attribute x_k. The standardized value is then calculated by subtracting the ratio of mean to standard deviation in the bin relative to x_j from each instance value in x_k. This approach is repeated for all attributes in the input space of vectors. The final step in the algorithm is to compute the max or the sum value of each data point. Large sum or max values indicate a high ratio of attribute noise compared to the other data points. Figure 1.6 shows the output from running the PANDA algorithm on the program example illustrated in Figure 4.4. The attribute noise column contains the attribute noise scores for each data point in the input data, whereas the remaining columns are the standardized values for each instance relative to other columns.



Line index

Comparisons between pairs of attribute values

Attribute Noise

Most Noisy

14	3.40	0.40	0.40	0.40	0.40	0.60	0.40	0.60	0.40	3.40	...	1.00	2.00	1.00	1.00	1.00	1.00	2.00	1.00	2.00	4.00
0	0.60	0.60	0.60	0.60	0.60	0.60	0.60	0.60	0.60	0.60	...	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00
1	0.60	0.60	0.60	0.60	0.60	0.60	0.60	0.60	0.60	0.60	...	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00
2	0.60	0.60	0.60	0.60	0.60	0.60	0.60	0.60	0.40	0.60	...	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00

Least Noisy

Figure 1.6: An ordered list of line vectors from most to least noisy.

1.1.7 Test Case Types and Code Change Categories

In this research, several dependency links between test case types and code change categories were identified. All of the test case types explored for dependency were extracted from the international standard ISO/IEC/IEEE CD 29119-1:2020 document [29], whereas the categories of code changes were

compiled from different studies in the literature. Table 5.2 presents definitions of 18 test case types and six categories of code changes that were investigated for a dependency in this research.

Table 1.2: The ISO/IEC/IEEE CD 29119-1:2020 definitions of test case types

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

A total of six categories of code changes could be identified from the literature. These are as follows:

[a] Memory management: This category of change is concerned with the management of the program memory during run-time. Examples of these memory related changes introducing memory leaks, buffer overflow, pointers' assignments, and resource interference when using multi-threading programming.

[b] Complexity: This category represents changes that affects the time

complexity of the program. It includes changes such as adding/removing loops, conditional statements, nesting blocks and/or recursions.

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

[c] Dependency: This category describes a code change that involves adding/ removing/ modifying a dependency to another module/ library. It can be importing/ removing/modifying a new library, a new namespace, or a new class.

[d] Conditional: This category of change occurs when a logical operator or a comparative value in a condition is modified. A misuse of logical expressions in conditional statements will affect the intended purpose of the program and will thus generate false outputs.

[e] Data change: This category involves 1) changing functions' parameters, 2) passing parameters of incompatible types to modules/functions, and 3) adding/fixing assignments of incompatible types to variables, casting statements, and array size allocations, and 4) modifying variable declarations

1.2 Related Work

Prior research on TCS can be broadly categorized into dynamic and static techniques [30]. This section highlights related work under these categories, and compare their effectiveness. Thereafter, we highlight related work that propose different class and attribute noise handling techniques for improving the learning of ML models.

1.2.1 Test Case Selection Approaches

Generally, TCS approaches search for dependencies between the modifications made to the source code and the set of affected test cases. Static analysis based approaches find these dependencies by constructing and traversing relational graphs in which the graph nodes represent classes, and connecting edges represent inheritance relationships [31].

Legunse et al. [30] implemented two static based test case selection techniques, one that collects dependencies on a file-level and another on a method-level. The tool parses the byte code of each changed file between two revisions and generates a class graph that captures dependency between multiple files. The tool takes two program revisions as input and the regression test suite. The output is a subset of tests from the input suite that s recommended for execution. Zhang [32] presented HyRTS, an approach that combines file and method granularity to analyze test dependency and change information. On a file-level granularity, the tool selects all the tests that cover file-level changes for execution; On a method-level granularity, the tool selects all the tests that overlap with the method-level changes for execution. Then, both sets of selected tests from the two granularity levels get merged and executed. Shi et

al. [31] conducted an experiment to investigate the safety and precision of three static analysis based techniques (Naive Analysis, String Analysis, and Border Analysis). The techniques showed an over-approximation of test dependencies, but a high safety in terms of not missing out affected tests.

Several techniques for TCS relied on information relating test cases to coverage of code [33] [34]. Rothermel et al. [33] investigated the use of statement (or branch) coverage to improve the rate of fault detection. Srivastava and Thiagarajan [34] proposed a prioritisation technique that is based on the changes that were made to the program, and focused on the objective function of block coverage to prioritize test cases. These techniques use greedy and genetic algorithms to gradually add prioritized test cases to an empty subset of test cases. The decision under which test cases get prioritized is based on the code coverage that they achieve. That is, the higher code coverage a test case achieves, the more prioritized it gets.

Other researchers employed similarity based test case selection to maximize test coverage, where the hypothesis is that *the more diverse test cases get, the higher their faults revealing capacity becomes.* To measure the diversity between test cases, similarity functions that compare different properties of test cases are employed. De Oliveira Neto et al [35] evaluated a similarity based technique, with respect to test coverage and time to execute the selected tests, on automated integration level testing. The similarity functions used in their study included Normalised Levenshtein, Jaccard Index and Normalised Compression Distance. Similarly, Hemmati et al. [36] employed a similarity-based TCS technique in the context of model-based testing. Their technique was designed to diversify elements in the test model, such as transitions and states that represent steps or conditions in abstract test cases.

Most of the existing techniques for TCS suffer from several drawbacks. First, a large number of these techniques use static code analysis, which does not scale well with large projects or higher level of testing, such as integration and system testing. Second, static analysis promotes for over-approximation of impacted tests, which means that a lot of unnecessary tests will still be executed (low precision) - resulting in limited success to reduce the cost of physical resources for testing. Third, using code coverage as the main criterion of interest for TCS leads to limiting conclusions about defects in the code, since no information about the detected faults is captured. Fourth, using code coverage based techniques requires maintaining dependency links between each test unit and test case identifiers, which is not feasible at large scale projects for regression testing. In the approach presented in this research, we aim to overcome these drawbacks by learning a ML model from data of historically committed code changes and impacted test cases. Unlike static analysis based techniques, our approach does not require parsing the code nor generating relational graphs to identify dependent files/methods that are affected by a change. Instead, the approach learns dependencies between impacted test cases and new code changes by analyzing the statistical count of previously occurring features (code constructs) that triggered a reaction among test cases.

1.2.2 Class and Attribute Noise Handling Approaches

Brodley et al. [18] uses an ensemble of classifiers, named Consensus Filter (CF), to identify and remove mislabeled instances. Using a majority voting mechanism with the support of several supervised learning algorithms, noisy instances are identified and removed from the training set. If the majority of the learning algorithms fail to correctly classify an instance, a tag is given to label the misclassified instance as noisy and later tossed out from analysis. The evaluation results show that when the class noise level is below 40%, filtering leads to better predictive accuracy than not filtering. On the basis of their experiments, the authors suggest that using any types of filtering strategies would improve the classification accuracy more than not filtering.

Guan et al. [17] introduced CFAUD, a variant of the approach proposed by Brodley et al. [18], which involves a semi-supervised classification step in the original approach to predict unlabeled instances. The approach was tested for an effect on learning for three ML algorithms (1-NN, Naive Bayes, and Decision Tree) using benchmark data-sets. The empirical results indicate that both majority voting and CFAUD have a positive effect on the learning of the three ML algorithms under four noise levels (10%, 20%, 30%, and 40%).

Muhlenbach et al. [37] introduced an outlier detection approach that uses neighbourhood graphs and cut edge weight algorithms to identify mislabeled data points. Instances identified as noisy are either removed or relabeled to the correct class value. Relabeling is done for instances whom neighbours are correctly labeled, whereas data points whom neighbouring classes are heterogeneously distributed get eliminated. The general data point drawn from the analysis showed that starting from 4% noise removal level and onward, using the filtering approach yielded better performance in 9 out of 10 of the domains data-sets.

Khoshgoftaar et al. [38] presented a rule-based approach that detects noisy data points using Boolean rules. data points that are detected as noisy are removed from the data before training. The approach was compared for efficiency and effectiveness against the C4.5 consensus filter algorithm presented in [18]. The results drawn from the case study suggests that when seeding noise in 1 to 11 attributes at two noise levels, the consensus filter outperforms the rule-based approach.

While the majority of the reported studies provide empirical evidence that support handling both class and attribute noise in data, the results from our research provide counter-evidence, opposing these findings when it comes to attribute noise. Our key data point from these counter-evidence results is in line with that described in [15], suggesting that the definition of noise is very much tied to the domain in which noise occur. In other words, handling attribute noise by identifying outliers in the attributes is observed not be harmful in the context of test case selection.

1.3 Research Design

This section describes the research focus of this thesis, the research methodology, and the data collection and analysis methods used in each of the appended papers.

1.3.1 Research Focus

The goal of this thesis is to create a method for selecting test cases that have the highest probability of revealing faults in the system under test, given new code changes pushed into the code-base. Figure 1.7 provides an overview of the appended papers in this thesis, and the order in which these papers were conducted. We began this research by designing an ML-based approach, called MeBoTS, (Paper A) for predicting and executing test cases given new code changes made by software developers. The conclusion drawn from this paper was that utilizing ML for TCS requires curating the training data by excluding large revisions and keeping smaller ones.

In Paper B, we focused on empirically examining the relationship between class noise, at different ratios, and the learning performance of the model in MeBoTS. The analysis showed a negative impact of class noise on the performance with respect to four evaluation measures (precision, recall, F1-score, and MCC). This called for designing a class noise handling algorithm to improve the performance of the ML model in MeBoTS (Paper C). Further, we examined the effect of attribute noise on the performance of MeBoTS in the same study presented in Paper C. The conclusion drawn from that study was that attribute noise is not necessarily harmful on the prediction of the model in MeBoTS.

In Paper D, we focused on identifying dependency links, by creating a taxonomy, between different types of code changes and test cases. The purpose was two-fold. First, we wanted to understand how to systematically map code changes and their relevant test case types in the training data of the ML model in MeBoTS. Second, we wanted to guide testers on deciding about which types of test cases should be executed during each CI cycle, given code changes of specific types. The conclusion from this paper was that testers should use performance related test cases to test code changes that are memory management and algorithmic complexity related.

The research questions in the included publications are as follows:

- 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? (Paper A).

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

- RQ3: How can we improve the predictive performance of a learner for test selection by handling class and attribute noise? (Paper C).

- RQ4: To which degree do software testers perceive content of a code commit and test case types as dependent? (Paper D).

We will now present the research methodology used to answer each of the research questions in the included publications.

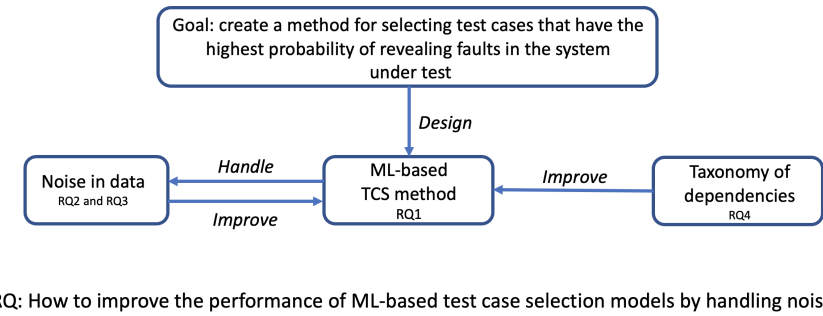


Figure 1.7: A summary of research focus.

1.3.2 Research Methodology

In this thesis, we address four research questions whose answers lead to answering the general research question. This section describes the research methodologies that we adopted to answer the four research questions of the appended papers in this thesis.

1.3.3 Using Textual Analysis and Machine Learning For Improving Test Case Selection (Paper A)

In Paper A, we chose to adopt the design science research as the methodology for answering the research question of *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?*. We followed the general three-steps process proposed by Runeson et al. [39], which consists of *problem conceptualization*, *solution design*, and *empirical validation*.

1.3.3.1 Problem Conceptualization

The problem of regression testing is widely discussed and studied in the literature. Most researchers in this field propose using test prioritization, minimization, or selection techniques to overcome the cost overhead in performing a re-test all approach for regression testing [40] [41] [42]. These techniques, however, suffer from a number of shortcomings that impede their usage, as pointed out in the related work section. Examples of the most commonly reported shortcomings include 1) high computational cost for parsing the code and generating relational graphs, and 2) imprecise approximation of the set of impacted tests, given new code changes.

Therefore, in Paper A, we set out to design and evaluate a new test case selection technique that utilizes machine learning and textual analysis for reducing the number of executed test cases in regression suites.

1.3.3.2 Design Artifact

We designed MeBoTS, a machine learning based approach, that is language-agnostic and does not require the code being analysed to be parsed. The method was inspired from a previous study conducted by Knauss et el. [3], where the authors used a statistical model to explore the relationship between historical code changes and test execution results. MeBoTS expands on this approach by utilizing textual analysis to extract code tokens from previously integrated code changes made in the main development branch, and uses those tokens as predictors for predicting the execution results (Pass and Fail) of test cases. MeBoTS treats code tokens as features and represents a code line with respect to its tokens' frequencies. To our knowledge, this way of extracting feature vectors from the source code is new, compared with other popular approaches for test selection. Unlike static code analysis techniques, MeBoTS can directly recognize what is written in the code without the need of parsing the code and accessing its abstract syntax tree for generating feature vectors.

Figure 1.8 provides an overview of the steps that comprise MeBoTS. A summary of each step is as follows:

[a] Step 1: The first step in the method is to collect a sample of previously committed code changes into the development repository as well as the respective set of test cases that was executed against the integrated code. For each pair of consecutive revisions from the drawn sample of revisions, all lines of code that were either added or modified are extracted and stored in a corpora of code changes. These lines are then assigned a label value that corresponds to the execution result of a test case that gets selected randomly from the extracted set of test cases.

[b] Step 2: The second step in the method is to extract feature vectors from the cumulative corpora of code changes (extracted in step 1) via a textual analysis technique. MeBoTS uses an open source tool called CCFlex [43] that utilizes the bag of words (BoW) model for performing features extraction (more details about how CCFlex works can be found in Paper A).

[c] Step 3: The final step in the method is to exploit the set of extracted feature vectors provided by the textual analyzer in step 2 for learning an ML model on classifying lines of code into either triggering a test case failure or pass. The goal is to form a generalization from the set of extracted training data that can be used to predict test execution result for newly pushed code changes and accordingly select impacted test cases for execution in the regression suite.

To better illustrate how the training data of MeBoTS gets constructed, we provide an example for a program written in the C++ language. Figure 4.2 illustrates two code fragments that belong to two revisions (revision 1 and revision 2). The framed line in revision 2 highlights the modified line in the program, relative to revision 1. Consider a scenario where we decide to use the execution result of test case `testisOdd`, shown in Figure 4.2, to guide the labelling of the modified line in revision 2. Then, a class value of '0' would be used to annotate a test failure, and a class value of '1' to annotate a test pass.

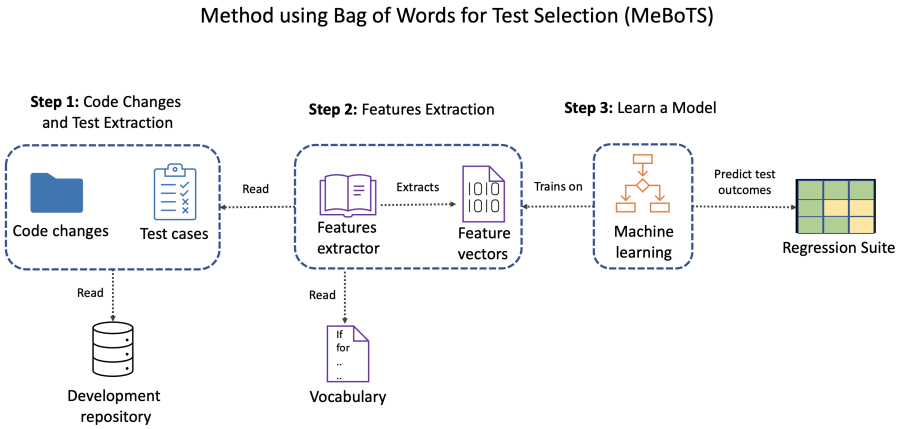


Figure 1.8: An overview of the MeBoTS method for test case selection.

Since test case `testisOdd` revealed no faults in revision 2 (i.e., the test passed), then a value of '1' is assigned to the modified line in the revision before it gets added to the training data of MeBoTS.

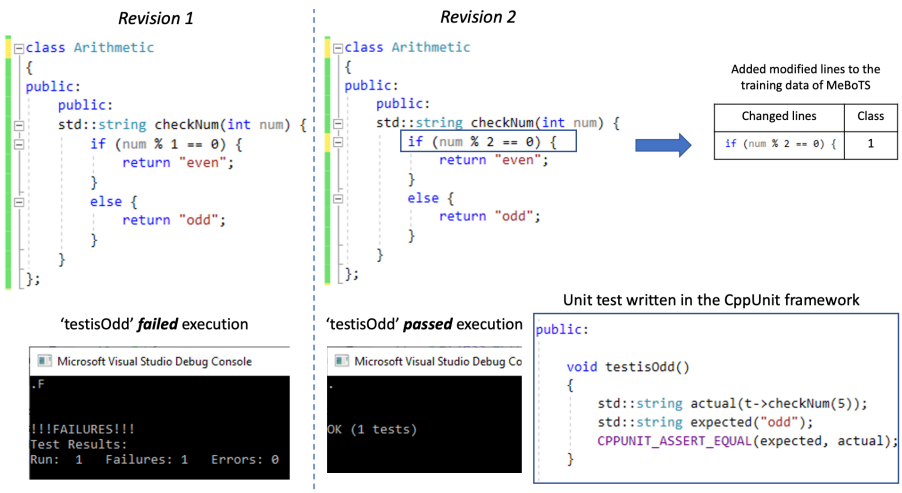


Figure 1.9: An example of how the training data for MeBoTS is constructed.

1.3.3.3 Empirical Validation

We evaluated the effectiveness of MeBoTS using an industrial data-set that belonged to a legacy system written in the C language. The collected data-set used for evaluation comprised of 82 code revisions that were integrated into the main development branch and verified during the build creation phase in the CI pipeline. Two evaluation trials of the ML model in MeBoTS were carried out. In the first trial, a total of 1.4 million lines of code and 82 test execution results were used for training and testing five different ML models -

three tree-based models and two deep learning networks. In the second trial, we chose to train and test the ML models on code check-ins that contained less than 100 thousand lines of code. The evaluation was done by measuring two performance metrics: precision and recall. We chose these metrics since they measure the model's ability to predict unimpacted test cases by new code changes. Particularly, recall measures the model's ability in identifying tests that require no execution, whereas precision would indicate the correctness of predictions with respect to the tests that were identified as unimpacted.

Since the goal of this study is to reduce the amount of test executions without altering the effectiveness of testing, we need to minimize the risk of missing out impacted tests in the regression suite. Therefore, we can accept some false alarms in the prediction of impacted tests, at the cost of running extra test cases over-nightly or weekly regression suites. Accordingly, we believe that the measure of precision is more important than recall in the context of this research.

After the evaluation of MeBoTS was completed, we inspected the type of source files (e.g., '.h', '.c', '.xml') in which the collected lines were modified/added. This was done as a sanity check to find out whether the evaluation of the ML models was based on lines that belonged to specific file types. The key observation from this inspection was that a lot of duplicate lines in the corpora appeared with different class labels. This data point called for further investigation to understand the impact of those duplicate lines on the predictive performance of the ML models in MeBoTS.

1.3.4 The Effect of Class Noise on Test Case Selection (Paper B)

The goal of the study presented in Paper B was to understand the impact of class noise in data-sets of code changes on the predictive performance of the ML model in MeBoTS. To answer the research question of *Is there a statistical difference in predictive performance for a test case selection ML model in the presence and absence of class noise?*, we designed and implemented a controlled experiment where we examined possible causality between class noise and the performance of the ML model in MeBoTS.

1.3.4.1 Goal Definition

The fact that two identical lines of code can be labelled differently - as pass or fail, is known in the literature as the class noise problem. In the context of this research, it can occur that duplicate lines of code are labeled with different test execution results for the same test. Such occurrences of *duplicate lines* in the training data become noise when fed as input to an ML model. While the problem of class noise is adequately discussed in the literature [16], [23], [44], there is a lack of studies that discuss its effect in the context of test case selection. The goal of this experiment was two-fold: 1) to understand the impact of different class noise ratios on the performance of the ML model in MeBoTS, and 2) to provide testers with an understanding on how to handle class noise before training a ML model for test selection.

1.3.4.2 Experimental Design

Following the guidelines proposed by Juristo et al. [45] for conducting controlled experiments in software engineering, we began the experiment by hypothesizing that class noise has a detrimental effect on the performance of the ML model in MeBoTS. We then broke down this hypothesis into more concrete hypotheses that specify the effect of class noise (independent variable) on four different performance evaluation measures (dependent variables), namely precision, recall, F1-score, and Matthew Correlation Coefficient (MCC).

The F1-score measure is a harmonic mean between precision and recall, indicating the exactness and completeness of the predictions made by the ML model. A drawback in using F1-score as a standalone metric is the fact that it only accounts for three elements in the confusion matrix (true positives, false positives, and false negatives). This can generate misleading conclusions about the performance of the model, particularly when the distribution of the binary classes in the data is imbalanced, as pointed out by Shepperd et al. [46]. Further, since the F1-score is only based on three elements in the confusion matrix, the count of true negatives is left unknowable. In our case, true negatives correspond to tests that are correctly predicted by the model to be impacted, and thus require execution. Software developers and other key stakeholders in a software project will probably appreciate knowing the number of correctly failing tests, given new code changes. Thus, neglecting the count of true negatives leads to missing valuable insights into the rate of faults in the source code. To mitigate these drawbacks that suffice in F1-score, we decided to measure the model's MCC, which takes into account the four elements in the confusion matrix. The closer the MCC score gets to 1, the higher the performance of the model is deemed. Measuring both the F1-score and MCC values would help us to increase confidence in the evaluation results. If both values showed different signs (positive and negative) then our evaluation to the model's performance will be different, depending on which performance metric we will choose.

To aid the hypotheses testing, a control group with 0% ratio of class noise was designed from the data-set presented in Paper A. This group was needed to build a baseline measure for comparing the learning performance of a model that gets trained on data without class noise.

1.3.4.3 Execution

Six variations of class noise (treatment levels) were selected in the experiment (10%, 20%, 30%, 40%, 50%, and 60%) to examine the effect of class noise on the four evaluation measures. We began by applying 15-fold stratified cross validation on the control group to generate experimental subjects into which the treatment was seeded. Each subject was treated as a hold out group for validating the ML model in MeBoTS after being trained on the remaining fourteen subjects. After seeding each treatment level into the experimental subjects, we fed those into a random forest ML model for training and evaluated its performance with respect to the four evaluation measures using the hold-out subject.

1.3.4.4 Hypotheses Testing

Finally, we tested the hypotheses of *there is a statistically significant difference in the performance of a test selection model in the presence and absence of class noise* using the Mann-Whitney and Kruskal-Wallis inference tests. The results revealed a statistically significant difference in the ML model's performance when being trained on a data-set with 0% class noise and with six variations of the treatment.

1.3.5 Improving Test Case Selection By Handling Class and Attribute Noise (Paper C)

In Paper C, we addressed the question of *how can we improve the predictive performance of a learner for test selection by handling class and attribute noise?* by carrying out a controlled experiment and a DSR methodology. The controlled experiment was designed to examine the effect of attribute noise removal on the performance of the ML model in MeBoTS, whereas the DSR was adopted to evaluate the effectiveness of a class noise handling algorithm in improving the performance of the ML model in MeBoTS.

1.3.5.1 Controlled Experiment

We followed the guidelines proposed by Juristo et al. [45] to design and implement the controlled experiment.

Goal Definition According to statistical literature, the presence of outliers in the data can introduce skewed results that lead to creating less precise ML models and a prolonged time of training [47], [48]. Therefore, a lot of studies that concern handling outliers has been proposed in the ML and statistical literature [49], [50], [51] .

Attribute noise occurs in training data as a result of selecting attributes that are irrelevant for characterizing the training instances. In the domain of test selection, such instances may occur when, for example, a small fraction of the overall number of conditional statements in the analyzed code deviates in syntax from the majority of conditional statements in the code. These deviations result in generating different feature vectors for similar lines, which may influence the performance of the ML model in MeBoTs. Therefore, the goal of the controlled experiment presented in Paper C was to examine the effect of removing lines of code that come with high attribute noise on the performance of the ML model in MeBoTs.

Experimental Design We began the design of the experiment by assuming that attribute noise removal can increase the learning performance of the test selection ML model in MeBoTS. To that effect, we used an existing attribute noise technique from the literature, called PANDA [25], which allows its users to remove as many noisy data points as desired from the training data before building a learning model. The PANDA algorithm recognizes training data points with large deviations from normal by comparing all pairs of attributes that were extracted from the training data. In total, four concrete hypotheses

were defined according to the goal definition. The hypotheses assume a positive effect of attribute noise removal on the ML model's precision, recall, F1-score, and MCC.

Execution The first experimental operation in this study was to implement the PANDA algorithm and to run it against the same data used in the study of Paper B. To examine the effect of attribute noise removal on the four evaluation measures, ten treatment levels were selected (5%, 10%, 15%, 20%, 25%, 30%, 35%, 40%, 45%, and 50%). Each treatment level corresponds to a fraction of training data points that gets removed from the training data before building the ML model in MeBoTS. In this experiment, we used 25-fold stratified cross validation to derive 25 experimental subjects on which the treatment was applied.

Hypotheses Testing Similar to the class noise experiment presented in Paper B, the Mann-Whitney and Kruskal-Wallis statistical test methods were performed to validate the hypothesis of *whether attribute noise removal has a statistically significant effect on the performance of a model for test selection.* The Mann–Whitney test was selected to perform a pairwise comparison between the evaluation measures under each treatment level and the same measures with no treatment (0% noise removal).

1.3.5.2 Design science research

The second part of the study presented in Paper C was concerned with designing a new class noise handling approach for improving the performance of the ML model in MeBoTS. For this purpose, we adopted the DSR methodology, following the same guidelines proposed by Runeson et al. [39], to evaluate the effectiveness of the algorithm in the context of test case selection.

Problem conceptualization The results from both experiments presented in Papers B and C provided us with insights into the effect of class and attribute noise in the context of test selection. Therefore, since class noise was identified as the only impeding type of noise on the classification performance of the ML model in MeBoTS, we decided to design and implement a tool that relies on domain knowledge for fixing the class value of training data points that are contradictory. The design of our algorithm is motivated by the low computational cost required for handling class noise, compared with existing approaches that use machine learning for identifying data points with class noise, such as majority voting [52] or entropy measurements [53].

Design artifact The algorithm starts by sequentially assigning a unique 8-digit hash value for each line of code in the original data set and creating an empty dictionary for storing unfiltered lines of code. After that, it iterates through the set of hashed lines in the original data set and saves non-repeated (syntactically unique) lines of code in the dictionary. Finally, the algorithm compares the class labels of each pair of `duplicate lines` in the original and dictionary sets. If the class label in the original set is labelled with 1 (passed) and the class label of the same instance in the dictionary is labelled

with 0 (failed), then the algorithm would relabel the class label of the line in the dictionary from 0 to 1. If the class values of both `duplicate lines` are assigned a class label of 1, then the algorithm would skip adding the line from the original set into the dictionary.

This way of handling class noise can be seen as both corrective and eliminating, since it 1) corrects the label of duplicate data points that first appear as failing and then pass the test execution, and 2) removes duplicate lines that are labeled as passing. Defective lines (labelled with 0) often occupy a small proportion of the overall fragment of code changes. Thus, a random line from a fragment, which was overall labeled as failing is more likely not to be the cause of the failure. Therefore, a design decision in the construction of the algorithm was to relabel lines as 'passed', if they have already been seen as part of non-failing fragments before. Thus, we select a more conservative approach when it comes to labeling lines as failing, in order to minimize the likelihood of mislabeling training data points.

Empirical Validation As soon as we finished from implementing the algorithm, we evaluated its effectiveness on the performance of the ML model in MeBoTS in the following manner: Firstly, we ran the algorithm against the original data-set (presented in Paper A), which resulted in a reduced data-set of 140,130 lines of code. Secondly, we trained the model in MeBoTS on both the original data and the class-noise handled version of the data respectively. Lastly, we compared the learning performance of the two models with respect to precision, recall, and F1-score.

1.3.6 Improving Test Case Selection By Creating a Dependency Taxonomy (Paper D)

In Paper D, we adopted a design science research methodology to answer the question of *To which degree do software testers perceive content of a code commit and test case types as dependent?* The goal of this paper was to design a taxonomy of dependencies between different test case types and code change categories. To aid the creation of the taxonomy, a combination of qualitative and quantitative methods were carried out, including literature review, survey, and a workshop with testers.

1.3.6.1 Problem Conceptualization

MeBoTS learns test case selection from training examples that include a mapping between code changes and execution results of test cases that were executed against the respective code. Since the current way of mapping occurs without a prior knowledge on what lines in the code triggered a test case reaction (to fail or pass), we risk introducing wrong training examples that do not characterize the underlying assumption of dependency. Therefore, in Paper D, we set out to create a faceted taxonomy (the design artifact) of dependency patterns between code changes and test case types to facilitate the construction of valid mappings in the training data of MeBoTS.

1.3.6.2 Design Artifact

To guarantee a systematic creation of the taxonomy, we followed the framework proposed by Usman et al. [54], which consists of four steps: planning, identification and extraction, design and construction, and validation.

Identification and extraction This phase is concerned with identifying the main categories that will be used in the taxonomy. The main categories correspond to types of code changes and test cases. To identify the categories and terms, we started by analyzing papers from the literature which discussed discussed a dependency pattern between code changes and test case types. In total, six categories of code change and 18 test case types were extracted. Further, 21 dependency links between the six categories of code changes and eight out of the 18 extracted test case types were identified.

Design and Construction In this phase, we utilized the extracted categories from the literature to develop a survey. The aim of the survey was to collect the opinions of industry experts from the testing community about possible dependency patterns between the extracted categories. The survey requested the invitees to provide a mapping between the categories extracted in the previous phase, where a single mapping corresponds to a dependency. In total, we received nine responses from testers who worked at three software development companies. The analysis of the survey responses showed that the strongest dependency patterns from the viewpoint of testers were concentrated around two categories of code changes and twelve test case types.

Due to the high level of agreement among the surveyed testers on the dependency between these two categories of code changes and the twelve test types, we decided to hold a workshop with testers to gain a deeper understanding about the connections and their strengths. Three out of the nine testers, who participated in the survey, and three other testers from another software company attended the workshop. During the workshop, the entire group of the six testers were asked to provide a ranking score (in a scale from 1 to 5) that reflects the dependency strengths between the two categories of code changes and the test types, where 1 in the scale corresponds to the least sensitive and 5 corresponds the highest sensitive. After discussing the sensitivity strengths, we asked the testers to justify their views about the sensitivity of each dependency by providing explanations for their ranking.

1.3.6.3 Empirical Validation

We validated our taxonomy by demonstrating and discussing the orthogonality between strongly dependent categories based on the input given by testers during the workshop. The demonstration offers a discussion about unique classifications that highlights types of test cases that are in relations with two categories of code changes. Nevertheless, we are aware of the importance of validating the taxonomy using utility demonstration or benchmarking, since both methods apply the knowledge gained on software artefacts, allowing researchers to empirically validate the taxonomy.

1.4 Summary of the Findings

The main goal of this thesis is to effectively select test cases that have the highest probability of revealing faults in the source code. This section summarizes the main findings drawn from each of the four appended papers in this thesis.

1.4.1 Predicting Test Case Verdicts Using Textual Analysis of Committed Code Churns (Paper A)

To find which test cases should be selected and executed in regression suites, we designed MeBoTS, a machine learning based method that learns the execution results of test cases by analyzing their relation with historically integrated code commits during CI. In Paper A, we evaluate the effectiveness of MeBoTS in selecting test cases using an industrial data-set of historical code changes and test execution results. In the following, we present two findings that were drawn from the study presented in Paper A.

1.4.1.1 Homogeneous and small revisions

The first finding is concerned with whether the size of code changes extracted from program revisions can affect the learning performance of the model in MeBoTS. The evaluation results showed that using a mix of large and small revisions for training the model in MeBoTS leads to producing random predictions of test execution outcomes (precision of 55% and 17.4% recall). On the other hand, the predictive performance of the same model was found to improve when using revisions that contain less than 100 thousand lines of code for training (73% precision and 49% recall). These validation results suggest that using MeBoTS allows testers to reduce the size of test suites by excluding 7 out of the 10 predicted passing test cases. However, there is a probability of 30% that we miss a test case failure, which means that the reduction of the test suite comes with a cost. This cost can be reduced, for example, if we collect the test cases that were not executed and execute them with lower frequency during a nightly or weekly test suite instead of hourly builds. Similarly, the measurement of the model's recall revealed an improvement in its ability to identify unimpacted tests by 32%, when being trained on smaller revisions. Based on this finding, we conclude that using a training data of smaller code revisions yields to building more effective model for test case selection.

1.4.1.2 The choice of the ML model in MeBoTS

The second finding from paper A is concerned with the suitability of different types machine learning models in MeBoTS. Namely, we compared the effectiveness of three conventional tree-based and two deep learning models in predicting test case execution outcomes. We found that the learning performance of the five models was almost similar. Specifically, the precision scores attained by the models ranged from 67% to 71%, whereas the recall scores ranged from 36% to 49%. Despite the similar performance scores produced by both tree-based and deep learning models, we argue that using a tree-based model in MeBoTS is a more appropriate choice due to the following reasons:

- tree-based models require lower computational cost compared with deep learning models
- tree-based models have a white-box nature, which allows testers/developers to gain access to the feature importance charts and thus trace and avoid pushing faulty patterns in future commits

All the papers included in this thesis contribute in finding an answer to the research question in Paper A. However, the findings drawn from Paper A formed the foundation to understand the suitability of different machine learning algorithms in MeBoTS and the effect of building a training data using a mix of large and small code changes on the effectiveness of MeBoTS.

1.4.2 The Effect of Class Noise On Test Case Selection (Paper B)

Class noise is the ratio of contradictory data points to the total number of points in the data-set at hand. Based on this definition, we designed and implemented a controlled experiment to empirically identify the effect of different ratios of class noise on the performance of the ML model in MeBoTS. In Paper B, we present the design and implementation of the controlled experiment. In the following, two findings from the controlled experiment are described.

1.4.2.1 Class noise decreases the predictive performance for test selection

The hypotheses testing results reported in Paper B revealed an inverse causality relationship between class noise and the performance of the model in MeBoTS. That is, as the noise level increased in the data, a decrease in the performance of the ML model was recorded. By seeding six treatment level of class noise (10%, 20%, 30%, 40%, 50%, and 60%), we statistically compared the effect of each noise level on the precision, recall, F1-score, and MCC of the model in MeBoTS. We found a significant difference between the precision, F1-score, and MCC scores under all the levels of class noise. Conversely, a significant difference between the recall scores was found only when the ratio of class noise exceeded 20%. The concordance between the F1-score and MCC metrics provided us with more confidence in the evaluation results, since both metrics recorded a negative trend when the level of class noise increased in the data. These findings suggest that the performance of the model in MeBoTS is hindered by the increase in the ratio of class noise.

The findings drawn from this experiment helped us to answer RQ2, which was positive (i.e., there is a statistical difference in the performance of the model in MeBoTS when class noise is present and absent from the data). The strong empirical evidence about a relation between class noise and the performance measures of the model in MeBoTS raised our awareness to the importance of using a class noise handling strategy before building the model in MeBoTS. Therefore, in the next paper, we introduce a noise handling strategy and evaluate its effectiveness on the performance of the model in MeBoTS. Further, in the same paper, we perform a controlled experiment to examine the effect of attribute noise removal on the data.

1.4.3 Improving Test Case Selection By Handling Class and Attribute Noise (Paper C)

In Paper C, we examine the effect of attribute noise removal on the performance of MeBoTS by carrying out a controlled experiment. Then, we present and evaluate the effectiveness of a class noise handling strategy that relies on domain knowledge for removing and relabelling contradictory data points in the training data. Lastly, we compare the effectiveness of the model in MeBoTS under three training trials: 1) using data with class and attribute noise, 2) using data with reduced class noise, and 3) using data with reduced class and attribute noise.

1.4.3.1 Handling attribute noise is not necessary

Our first finding suggests that removing data points with high attribute noise from the training data has no statistically significant impact on the learning of the model. Counter-intuitively, we found that training the ML model after removing 20% of data points with the highest attribute noise ratio is penalized by a 3% decrease in the model's precision and an 8% decrease in recall. As such, the removal of data points that come with attribute noise implies information loss and thus a higher risk of missing faults in the code. Therefore, testers are recommended not to account for handling attribute noise in the data using an elimination based approach.

1.4.3.2 Using domain knowledge for handling class noise is effective

We found that handling class noise in the data using a domain knowledge based strategy improves the predictive performance of the ML model in MeBoTS. Using the noise handling strategy presented in Paper C allows testers to correctly exclude 8 out of 10 actually passing test cases from execution (precision = 81%). Further, using the noise handling strategy results in a 70% reduction in the amount of false negatives (recall improvement from 17% to 87%), which means that the probability of missing out passing tests is reduced by 70%.

In summary, the findings drawn from Paper C provide empirical evidence that support the use of a class noise handling strategy to improve the predictive performance of the model in MeBoTS. Further, they counter-act exhaustive noise handling efforts by showing that attribute noise removal is not necessarily important for improving the performance of the model in MeBoTS. To further improve the effectiveness of MeBoTS in test selection, we conducted the study presented in Paper D.

1.4.4 Improving Test Case Selection By Creating a Dependency Taxonomy (Paper D)

In Paper D, the aim was to improve the effectiveness of MeBoTS by identifying dependency links between code changes and test case types. The identification of these links can guide the creation of training data that correctly maps the analyzed code with the execution results of dependent test case types.

1.4.4.1 Performance tests are strongly dependent on memory and complexity changes

During the workshop conducted in the study presented in Paper D, testers were mainly focused on software quality aspects such as performance and security when discussing the dependencies between different test case types and memory and algorithmic complexity changes. The identified dependency patterns in Paper D suggest that the performance of the ML model in MeBoTS can be improved by mapping memory management code changes with the execution results of either performance, capacity, load, stress, soak, and volume test cases. Similarly, complexity code changes were found to be in dependency with the same test case types in addition to maintainability test cases.

1.4.4.2 Mixed views on security tests

We found a wide disparity in the views of testers regarding the sensitivity of this test type. 33% of the testers perceived this test type to be sensitive to memory changes, 17% perceived it to be somewhat sensitive, whereas 50% of testers perceive a low sensitivity to memory changes. Testers who considered this test type to be sensitive argued that memory changes lead to memory leaks which, if not properly managed, might expose the system to security breaches. Disagreeing participants argued that resource leaks result in performance issues rather than security breeches. Further, they linked the sensitivity of security tests to the program domain, as being a strong determining factor for establishing dependency links. This lack of consensus among testers about security testing on the one hand and memory and complexity changes on the other hand calls for future work that empirically investigates the causality between code changes and security tests.

1.5 Discussion

The results of this thesis show that utilizing textual analysis and machine learning for test case selection has a promising potential in reducing the size of regression test suites. The value of the presented approach lies in being language agnostic, and not in need to parse the code being tested. The ultimate goal of this approach is to improve the effectiveness of performing regression testing during build suites in CI, and not to abolish practices of exhaustive testing that companies perform over nights or on a weekly basis.

In our quest to improve the performance of the ML-based method, we found that class noise has an impeding effect on the learning of the model, resulting in an over-approximation of tests that require execution (lower F1-score and MCC). Hence, a first step towards improving the effectiveness of MeBoTS is to handle class noise in the data before learning a model for TCS. Generally, the appropriateness of which strategy to use for handling class noise depends on the domain in which noise is incurred. In the domain of TCS, it is not trivial to define a general rule to identify which class label should be corrected/removed in the training data, since we do not know exactly which lines in the code are mislabelled and by what sources of noise. We cannot simply re-label any line, as suggested in the majority voting algorithm [52],

since we risk loosing information that can lead to drawing correct predictions about test case outcomes. Another factor that acts against the use of existing tools in the testing domain is the fact that they are computationally expensive. Since "time" is a crucial factor in determining the success of TCS tools, we need to aim at minimizing the use of tools that require high computational power, if possible. For these reasons, a lightweight tool that uses domain knowledge was designed and evaluated in this research. Whilst the designed tool showed a positive impact on the performance of the ML model in MeBoTS, it is important to note the following remarks. First, there is no guarantee that an even distribution of points in the noisy data-set will be maintained after the tool relabels contradictory entries. Thus, there is a possibility of either including or excluding fault revealing tests in the regression suites. Excluding more tests in the suites implies higher risks that defects remain undetected, whereas including more tests implies higher cost of testing. Based on this, it is recommended that testers ensure an even distribution of the classes after cleaning class noise from the data. Second, the corrections made by the tool assume that faulty lines always appear in less frequency than non-faulty ones, which means that the tool is susceptible to make erroneous labelling when faulty lines are labelled correctly in the original data. However, this risk is exhibited by the majority of existing ML-based correction techniques, such as majority voting and CFAUD [17] [18]), which require high computational cost. Therefore, we compromise some errors in mislabelling at the interest of saving time for testing.

Contrary to the finding that confirms the importance of handling class noise before learning a model, we could not draw a similar conclusion about the impact of removing attribute noise on the performance of the learning model in MeBoTS. In other words, removing lines that contain exceptional attribute values did not lead to any improvement in the learning of the ML model. The key observation from this finding is that the harmfulness of attribute noise is tightly coupled with the domain in which the noise is presented.

1.6 Threats to Validity

In this thesis, we analyze the threats to validity by relying on the framework recommended by Wohlin et al. [55] and discuss the validity in terms of external, internal, construct, and conclusion.

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?*

Small sample sizes. The data sets under which the appended papers in this thesis were conducted belonged to small sample sizes. Papers A, B, and C were based on the analysis of a sample that belonged to twelve test cases and 82 code revisions for a single industrial program. Paper D was based on the analysis of the opinions of a small sample of testers (nine testers in the survey and six testers in the workshop) about dependency between code changes and test case types. We are aware that using such a small sample of test cases and

and testers can hinder the generalization of the findings outside their context. However, we minimized this threats by randomly selecting both samples of test cases and testers such that the likelihood of drawing a representative sample increases.

1.6.2 Internal Validity

Internal validity concerns issues that may indicate a causal relationship between independent and dependent variables, although there is none.

Machine learning models. In the work presented in Papers B and C, we used a random forest ML model to evaluate the effectiveness of MeBoTS in the presence and absence of class and attribute noise. However, we did not experimentally assess the effectiveness of tuning the model's hyper-parameter on the predictive performance. Therefore, there is a likelihood that the evaluation results drawn in Papers B and C will differ when other configuration parameters are set for the model. However, in Paper C, we minimized this threat by running an additional trial of training where we evaluated the effect of changing the number of trees in the model from 100 to 300. The results showed that the learning performance of the model was similar when using 100 or 300 trees.

Another internal threat to the validity lies in the work presented in Paper D, which concerns the long waiting time between receiving the survey responses and the conduct of the workshop. During this time, testers who answered the survey and participated in the workshop may have changed their views about the dependency patterns. As a consequence, a misalignment in the dependency links provided by the testers in the survey and the workshop may have occurred. However, we minimized the impact of this threat by defining and explaining all the terms (code change categories and test case types) that were provided in the survey during the workshop. Another internal threat to validity is the likelihood that testers were influenced by the opinions of each other. However, since we construct our taxonomy based on a triangulated approach, we minimize the likelihood of such a risk.

1.6.3 Construct Validity

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

Nature of test case failures. In this research, it is difficult to determine whether the collected execution results of test cases are connected to faults in the code base and not due to external factors such as flaky tests, machinery failures, or an environment upgrade during test execution. This poses a threat to the construct validity since our research is concerned with finding a statistical dependency between changes in the code and test case execution results. However, this threat was minimized by randomly selecting test execution results from the pool of executed tests.

Majority class problem. After applying the treatment on the experimental subjects in Papers B and C, there is a chance that we imbalanced the distribution of the binary classes in the training data. This implies a risk of introducing a bias in the predictions towards one of the classes. Due to the high computational cost required to check the balance across all experimental subjects, we could

only verify that the distribution of the classes was balanced under 1 treatment level. However, the downward trend in the predictive performance of the four evaluation metrics, after seeding each level of noise, indicated no bias towards a particular class.

Assumption of dependency. This research is built on the assumption that there exists a dependency between code changes and test types. Nevertheless, there is a chance that such a dependency does not exist and that the dependencies reported in Paper D were derived coincidentally. We minimize this risk by constructing the taxonomy in Paper D from the viewpoints of practitioners.

1.6.4 Conclusion Validity

Conclusion validity focuses on the ability to draw correct conclusions about relations between the treatment and the outcome of our study. *Over-approximation of impacted test cases.* The biggest threat to the conclusion validity of this research lies in the fact that the ML model in MeBoTS over-approximated the number of impacted test cases. Particularly, in Paper A, the highest recall score attained by the ML model was at 49%, which implies that 51% of test cases that the model predicted to fail were actually false alarms and should not be executed in the regression suite. However, in Paper C, we mitigated this threat by handling class noise in the data, which resulted in constructing a less over-approximating model (recall at 87%).

Differences among subjects. The descriptive statistics in Papers B and C indicated that we have a few outliers in the performance measures obtained by the models across the experimental subjects. These outliers can impact the conclusions drawn from the analysis results. Therefore, we mitigated this risk by running two trials of analysis - with and without outliers - to check if the outliers had any impact on the results. Based on the analysis, we found that removing the outliers did not change the results. Thus, we decided to include the outliers in the analysis.

Bias in extracting terms from the literature. The identification and extraction of terms for the taxonomy building in Paper D was derived from the literature. However, this was not based on a systematic literature review guidelines, which introduces a risk of possible bias in the selection of code categories and test case types. However, we minimize this risk by inviting testers to propose other types of code changes and test cases that are not provided in the survey and the workshop.

1.7 Summary

Software test case selection is a broad area of active research that is needed to reduce the cost of testing and to increase the speed of products' delivery to the market. Finding an effective and an easy-to-maintain test case selection solution that can accommodate for the growing need of quick code integration and deployment during CI remains as an open challenge for researchers and practitioners.

This research set out to develop and investigate the effectiveness of a test selection method, called MeBoTS, that utilizes machine learning and textual analysis. By conducting a series of studies using both design science

and controlled experiment, we found potentially promising results about the applicability of the method in achieving an effective test case selection. In this research, we particularly focused on gaining empirical evidence on the importance of handling class and attribute noise in the training data before using MeBoTS. Additionally, we designed a taxonomy of dependencies between code changes and test case types to further improve the effectiveness of MeBoTS in learning test case selection. The findings drawn from this work support the necessity of handling class noise from the training data before using MeBoTS. In contrast, no statistical evidence to support the necessity for handling attribute noise could be found. Finally, the training data for MeBoTS should be done based on a systematic mapping between memory management code changes and performance, load, soak, stress, volume, and capacity tests. Similarly, the same types of tests, in addition to maintainability tests, should be used for mapping with complexity code changes.

1.8 Future Work

The long-term goal of this research is to integrate MeBoTS as a plug-in to existing IDEs, such as Visual Studio or Eclipse, ideally, to allow developers pinpoint exactly which lines in their code will trigger a test failure at pre-commit, and thereby to save companies large cost of testing. We believe that there are still several questions that need to be answered before realizing this vision in practical terms. However, the empirical results reported in this thesis provided us with an understanding of the challenges and future work that need to be carried out in the future. We now present a road map for future work that we plan to carry out.

Evaluate the effectiveness of MeBoTS in predicting unobserved code revisions: In this research work, the ML model in MeBoTS was built and validated on data-sets that were collected from the same code revisions. As a future work, we plan to validate the model on unseen code changes that belong to future revisions. An additional plan is to determine the amount of time required to retrain MeBoTS on new code revisions before its predictive performance deteriorates over time.

Evaluate the effectiveness of MeBoTS when trained on validated dependency patterns: The taxonomy presented in Paper D was validated by discussing the orthogonality between different categories of code changes and test types. A future work is to continue working on refining the taxonomy by investigating additional dependency links between code changes and test case types. Further, we aim at increasing the validity of the taxonomy by evaluating it on industrial software artefacts. After that, we plan to construct training data using the validated dependency links and evaluate potential learning improvements attained by the ML model in MeBoTS.

Evaluate the effectiveness of other attribute and class noise handling strategies in different contexts: A future work is to examine the versatility of different class and attribute noise handling strategies in improving the

prediction performance of ML models in solving different software engineering tasks, including code reviews and fault predictions.

Evaluate the effectiveness of MeBoTS using more data-sets and different programming languages: In this research, the effectiveness of MeBoTS was evaluated on an industrial data-set of code changes written in the C++ programming language. Future work aims at expanding the set of explored languages to include Java and Python software programs.

Compare the effectiveness of MeBoTS with existing test case selection techniques: Future directions aim at comparing the effectiveness and efficiency of MeBoTS with existing test case selection techniques, such as those that use static code analysis and heuristics-based algorithms.

Evaluate the effectiveness of MeBoTS when trained on different granularity levels: In this research, we evaluated the effectiveness of MeBoTS in predicting test case verdicts by carrying out a line-level analysis of code changes. Future work aims at exploring the effectiveness of MeBoTS on different levels of granularity, including method and class levels.

Evaluate the effectiveness of MeBoTS when trained on test scripts changes: This research examined the effectiveness of MeBoTS in predicting test case verdicts, given changes made to the code base. Future work aims at investigating the effectiveness of the method by analysing changes made to the test scripts. Additionally, the same analysis of test scripts is planned to be used for predicting flaky tests.

Chapter 2

Paper A

Predicting Test Case Verdicts Using Textual Analysis of Committed Code Churns

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

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

Abstract

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

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

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

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

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

2.1 Introduction

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

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

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

Our research is inspired from a previous study conducted by Knauss et el. [3], where the authors explored the relationship between historical code churns and test case executions using a statistical model. Their method used precision and recall metrics in predicting an optimal suite of functional regression tests that would trigger failure. In this paper we expand on that approach by going one step further – conducting a textual analysis of what is the code that is actually being integrated. For example, instead of using code location as the parameter, we use such measures as the number of 'if' statements or whether the code contains data definitions. Similarly, our choice of using code churns is inspired from the work of Nagappan and Ball [58], in which the authors presented a technique for early faults prediction using code churn measures. In their publication, the authors identified a positive correlation between the size of code churns and system defects density. Our method builds on this and uses the Bag of Words (BOW) approach to extract features from code churns. This enables the identification of statistical dependency between keywords and test case verdicts. For example, a churn containing a frequent occurrence of keywords like 'new' or 'delete' might trigger specific tests to fail. More precisely, we aim at investigating the following research question:

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

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

To address this research question, we conduct a design research study, where we develop a new method and evaluate it on the company's data set. Our method is based on the research by Ochodek et al. [43], which uses textual analyses to characterize source code. Our MeBoTS method builds on that by using historical test verdicts as predicted variables and uses Random Forest algorithms to make the predictions.

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

2.2 Background

2.2.1 Categories of Machine Learning

Machine learning is a class of Artificial Intelligence that provides systems the ability to automatically make inferences, given examples relevant to a task [59]. The main advantage of using Machine learning over classical statistical analysis, is its ability to deal with large and complex data-sets [60]. These systems can be classified into four categories depending on the type of supervision involved in training: a) Supervised, b) Unsupervised, c) Semi-supervised, d) Reinforcement Learning [59]. Since we view the problem of predicting test case verdicts as a classification problem, we briefly mention the supervised learning category.

In supervised learning the training data-set fed into the ML model contains the desired solution, called labels. The model tries to find a statistical structure between these examples and their desired solutions [60]. A typical task for this kind of learning is classification.

2.2.2 Tree-based and Deep Learning Models

In Machine learning, a decision tree is an algorithm that belongs to the family of supervised learning algorithms. The algorithm has an inherent tree-like structure and is commonly used for solving classification and regression problems [61]. Starting from the root node, the algorithm uses a binary recursive scheme to repeatedly split each node into two child nodes, where the root node has the complete training sample [62]. The resulting child nodes correspond to features in the training data, whereas the leaf nodes correspond to class labels (binary or multivariate). Other algorithms, such as Random Forest and Adaptive Boosting, use Decision trees as a primary component in their structure. These algorithms build a collection of decision trees, called an ensemble, to increase the overall learning of the classification or regression task at hand [60].

Deep Learning is a branch of ML that was founded on the premise of using successive learning "layers" to achieve more useful representation of the data [59]. The learning of these successive layers are achieved via models called Neural Networks (NN) [59]. A multilayer NN is one that consists of at least three layers: 1) one input layer, 2) at least one hidden layer, 3) and an output layer

of artificial neurons [63]. Similarly, a Convolutional network (CNN) consists of a set of learning layers [59]. The main difference between the two networks is in the way they search for patterns in the input space [60]. More precisely, a CNN works by sweeping a matrix-like window, called filter, over every location in every patch to extract patterns from the input data [60]. As opposed to Decision Trees, ANN have a black box nature, which means that no insight about how their predictions were made can be easily accessible [60]. Nevertheless, the main advantage of using deep learning comes from their ability to handle large and complex data-sets of features.

2.2.3 Code Churns

The amount of changes made to software over time is referred to as code churn [58]. As new churns are added, new risks of introducing defects into the system emerge [64]. According to Y. Shin et al. [64], each check-in made into a version control system includes newly added or deleted code that increase the chances of triggering failures. At some point in time, an evolving system may be vulnerable, on average, to one extra fault for every new additional change [65]. For example, in C programming, the declaration of 'static' local and global variables are among the most confused keywords by developers, as each static local and global declaration has a different effect on how the data will be retained in the program's memory [66].

2.3 Related Work

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

2.3.1 ML-based Test-Case Selection

Around 2015/16, we find the first machine learning based approaches for test-case selection. With only 4 studies included in the systematic mapping study by Durelli et al. [67], the use of machine learning for test case prioritization seems to be new.

Busjaeger and Xie [68] present an industrial case study in which a linear model is trained with the SVMmap algorithm using the features Java code coverage, text path similarity, text content similarity, failure history, and test age. The evaluation on the industrial case study, considering 2000 changes and over 45 000 test executions shows an Average Percentage of Faults Detected (APFD) of around 85%.

Chen et al. [69] prioritize test programs for compilers "identifies a set of features of test programs, trains a capability model to predict the probability of a new test program for triggering compiler bugs and a time model to predict the execution time of a test program."

Spieker et al. [70] introduced Retecs, a reinforcement learning-based approach to test case selection and prioritization. Retecs considers duration of a test case's execution, previous last execution and failure history. Online learning is used to improve test case selection between continuous integration cycles. The approach was evaluated on 3 industrial data sets, including together

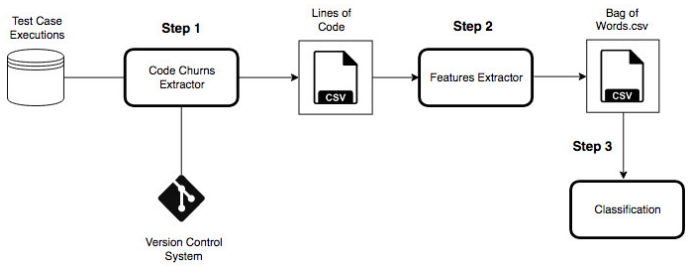


Figure 2.1: The MeBoTS method.

more than 1.2 million verdicts, and achieved a normalized Average Percentage of Faults Detected (APFD) of around 0.4 to 0.8 depending on the data set.

Most recently, Azizi and Do [71] perform test case prioritization by calculating a ranked list of components considering the access frequency of a component as well as a fault risk. The fault risk for each component is thereby predicted using a linear model of change and bug histories. Test cases associated with highly ranked components are prioritized. The approach was evaluated on three web-based systems and where it could reduce the number of test cases by 20% while still finding over 80% of the errors.

Palma et al. [72] replicate and extend a work of Noor and Hemmati [73] and [74], to predict test case failure based on a machine learned model basing on test quality metrics as well as similarity-based metrics.

However, to the best of our knowledge, no other learning-based method works for code-churns. The only exception is one of our previous collaboration with Knauss et al. [3]. The introduced code-churn based test selection method (CCTS) analyzes correlations between test-case failure and source code change. The approach was evaluated in several configurations, leading to results ranging from 26% precision up to a 54% with a 97% recall. We deem these results promising and one of the main motivations for this study.

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

The following section is a description of the MeBoTS method used in this research, which comprises of three sequential steps, as shown in Figure 2.1. The method utilizes two Python programs and an open source textual analyzer program, called CCFLEX [43].

2.4.1 Code Churns Extraction (Step 1)

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

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

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

Table 2.2: Input to the textual analysis and feature extraction

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

was executed, and the verdict value - as shown in Table 2.1.

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

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

2.4.2 Textual Analysis and Features Extraction (Step 2)

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

- creates a vocabulary for all lines (using the bag of words technique, with a specific cut-off parameter),
- for the words that are used seldom (i.e. fall outside of the frequency defined by the cut-off parameter of the bag of words), a token is created,
- finds a set of predefined keywords in each line,
- check each word in the line whether it is part of the vocabulary, it should be tokenized or if it is a predefined feature.

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

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

Table 2.3: Output from the feature extraction algorithm

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

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

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

Table 2.3 is a large array with the numbers, each representing the number of times a specific feature presents in the line. This way of extracting information about the source code is new in our approach, compared to the most common approaches of analyzing code churns. Compared to the other approaches, MeBoTS recognizes what is written in the code, without understanding of the syntax or semantics of the code. This means that we can analyze each line of code separately, without the need to compile the code or without the need to parse it. This means, that we can take code churns from different files in the same baseline and analyze them together. MeBoTS also goes beyond such approaches like Nagappan et al. [58], which characterizes churns in terms of metrics like number of churned lines or churn size.

2.4.3 Training and Applying the Classifier Algorithm (Step 3)

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

Table 2.4: Input to the Classifier Model

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

2.5 Research Design

2.5.1 Collaborating Company

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

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

2.5.2 Dataset

The data-set provided by company A contained historical test case execution results for a mature software product that has evolved for almost a decade. The analyzed product consisted of over 10 thousand test cases and several million lines of code written in the C language. We decided to test the MeBoTS on a set of randomly selected tests that, presumably, reacted to changes in the source code during different CI cycles. Our selection of test cases was based on the granularity of test executions whose verdicts changed from one state to another(see Table 2.1).

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

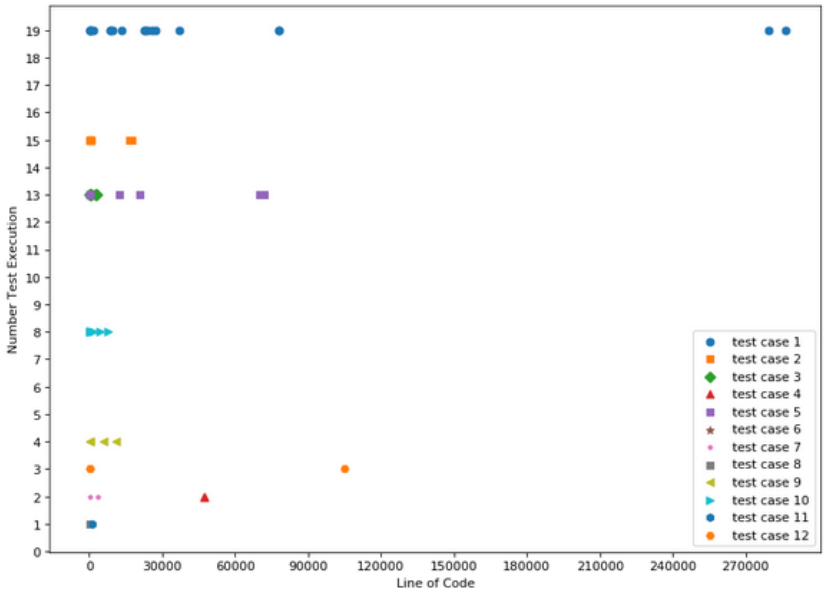


Figure 2.2: Churn Size per Test Execution Plot.

out tests that were executed on churns whose total size exceeded 110 thousand lines. The visual inspection of the curated data-set is represented in Figure 2.3, which comprised of 290 thousand lines of code, containing a fairly balanced representation of the binary classes (passed and failed), with 110 thousand lines belonging to the passed class and 180 thousand lines belonging to the failed class.

The two data-sets described above were used for training the ML models. The first data-set was used in the first phase, whereas the second curated data-set was used in the second phase of ML training, and ultimately became our focus due to data size homogeneity.

2.5.3 Evaluating and Selecting a Classification Model

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

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

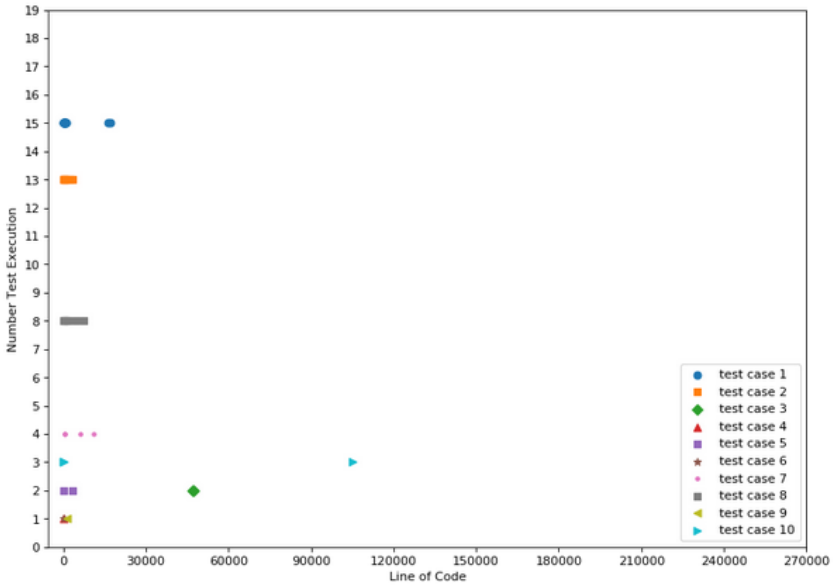


Figure 2.3: Churn Size per Test Execution After Curation Plot.

Table 2.5: The Evaluated Models and Their Hyper-Parameters

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

follow: 70% for the training set and 30% for the test set.

To save the long run time required by automated hyper-parameter tuning tools such as grid and random search, the configuration of the models was done manually. Table 2.5 summarizes the hyper-parameters used for training the models. We used the implementation of Decision Tree, Random Forest, and AdaBoost algorithms available in the Python scikit-learn library [75] and then used the Keras library [76] for the implementation of Multilayer NN and CNN.

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

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

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

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

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

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

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

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

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

2.6 Results

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

2.6.1 Training the Models on Churns of Varying Sizes

The evaluation of the five models in the first iteration reports a mean precision of 47% and a mean recall of 17%, suggesting that all models achieved low performance. The best result was obtained by the Multilayer NN model with a precision rate of 50.3% and a recall rate of 31%. The precision and recall rates for the five models can be seen in Figure 4. The interpretation of these values suggest that out of the 406,948 lines of code that actually triggered a test case failure, the model correctly predicted test failure for 226,994 lines,

Table 2.6: Models Evaluation before Data Curation

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

whereas it could correctly predict a passing test verdict for 5,758 out of 11,428 lines. Similarly, the results of the four categories of errors for the other models can be seen in Table 2.6 and interpreted in the same fashion.

2.6.2 Training the Models on Churns of Small Sizes

The second iteration of analysis involved training the same set of models on the curated data-set, which excluded tests covering churns with quantities above 110k lines of code. The results, shown in Figure 2.5, indicate an improvement in precision and recall when compared to the results in the first iteration for the same types of models. Table 2.7 reports the results from the second round of training, showing a mean precision of 70% and a mean recall of 44.5%. The Multilayer NN model performed best in prediction, such that, it correctly predicted 48,755 lines that actually triggered a test case failure, out of 67,363 lines in the test set.

Table 2.7: Models Evaluation after Data Curation.

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

2.6.3 Implication

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

2.7 Validity analysis

When analyzing the validity of our study, we used the framework recommended by Wohlin et al. [55]. We discuss the threats to validity in two categories: internal and external. Typically, a number of internal threats to validity emerge

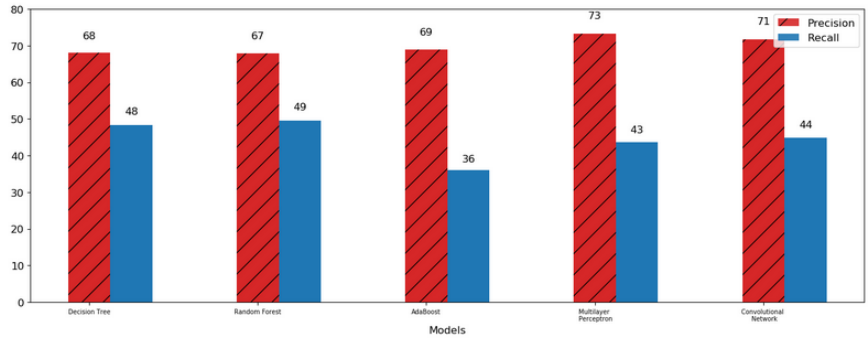


Figure 2.5: Precision and Recall of The Models After Data Curation

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

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

2.8 Recommendations

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

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

- We suggest to only utilize code churns that are homogeneous and small in

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

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

2.9 Conclusion and Future Work

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

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

Acknowledgment

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

Chapter 3

Paper B

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

Al-Sabbagh KW, Hebig R, Staron M.

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

Abstract

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

3.1 Introduction

In testing large systems, regression testing is performed to ensure that recent changes in a software program do not interfere with the functionality of the unchanged parts. Such type of testing is central for achieving continuous integration (CI), since it advocates for frequent testing and faster release of products to the end users' community. In the context of CI, the number of test cases increases dramatically as commits get integrated and tested several times every hour. A testing system is therefore deployed to reduce the size of suites by selecting a subset of test cases that are relevant to the committed code. Over the recent years, a surge of interest among practitioners has evolved to utilize machine learning (ML) to support continuous test case selection (TCS) and to automate testing activities [77], [78], [79]. Those interests materialized in approaches that use data-sets of historical defects for training ML models to classify source code as defective or not (i.e. in need for testing) or to predict test case verdicts [78], [80], [77].

One challenge in using such learning models for TCS lies in the quality of the training data, which often comes with noise. The ML literature categorized noise into two types: attribute and class noise [81], [82], [26]. Attribute noise refers to corruptions in the feature values of instances in a data-set. Examples include: missing and incomplete feature values [83]. Class noise, on the other hand, occurs as a result of either contradictory examples (the same entry appears more than once and is labeled with a different class value) or misclassification (instances labeled with different classes) [84]. This type of noise is self-evident when, for example, analyzing the impact of code changes on test execution results. It can occur that identical lines are labeled with different test outcomes for the same test. These *identical lines* become noise when fed as input to a learning model.

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

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

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

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

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

3.2 Definition and Example of class Noise in Source Code

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

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

Since the contradictory entry can only be among two (or more) entries, the number of all entries for which a duplicate entry exists with a different class label. A duplicate entry is an entry that has the same line vector, but can have different labels. For example, a data-set containing ten duplicate vectors with nine that are labeled `true` and one labeled `false` has ten contradictory entries. It is not trivial to define a general rule to identify which class label is correct based on the number of entries. For example, noise sources might systematically tend to introduce false "false" labels. Since we do not know exactly which class should be used in this context, we cannot simply re-label any instance, as suggested by the currently used solutions (e.g. using majority voting [52] or entropy measurements [53]) and therefore we count all such entries as contradictory. As an illustration of the problem, in the domain of TCS, Figure 3.1 shows how a program is transformed into a line vector and assigned a class label. It illustrates how a data-set is created for a classification task to predict whether lines of a C++ program trigger a test case failure (class 0) or a test case pass (class 1). The class label for each line vector is determined by the outcome of executing a single test case that was run against the committed code fragment in CI. In this study, a class value of '0' annotates a test failure, whereas a class value of '1' annotates a passed test. The Figure shows the actual code fragment and its equivalent line vector representation achieved via a statistical count approach (bag-of-words). The line vectors in this example correspond to source code tokens found in the code fragment. Note how lines 5 and 11 are included in the vector representations, since brackets are associated with loop blocks and function declarations, which can be important predictors

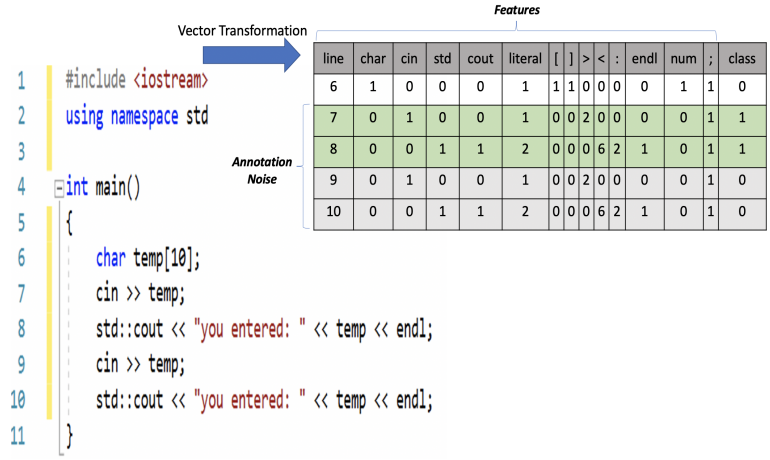


Figure 3.1: Class Noise in Code Base.

to capture defective patterns. All shaded vectors in the sparse matrix (lines 7 to 10) are class noise since pairs (7,9) and (8,10) have the same line vectors, but different label class – 1 and 0. The green shaded vectors are 'true labeled instances' whereas the gray shaded vectors are 'false labeled instances'. Note that the Table in Figure 3.1 shows an excerpt of the entries for this example. Since there are 11 lines of code, the total number of entries is 11. The formula for calculating the noise ratio for this example is thus:

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

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

3.3 Related Work

Several studies have been made to identify the effect of class noise on the learning of ML models in several domains [16], [23], [24]. To our knowledge, no study addresses the effect of class noise on the performance of ML models in a software engineering context. Therefore, understanding the impact of class noise in a software engineering context, such as testing, is important to utilize its application and improve its reliability. This section presents studies that

highlight the impact of class noise on performances of learners in a variety of domains. It also mentions studies that use text mining and ML for TCS and defect prediction.

3.3.1 The Impact of Noise on Classification Performance

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

Nettletonet et al. [16] examined the impact of class noise on classification of four types of classifiers: naive Bayes, decision trees, k-Nearest Neighbors, and support vector machines. The mean precision of the four models were compared under two levels of noise: 10% and 50%. The results of the comparison showed a minor impact on precision at 10% noise ratio and a larger impact at 50%. In particular, the precision obtained by the Naive Bayes classifier was 67.59% under 50% noise ratio compared with 17.42% precision for the SVM classifier. Similarly, Zhang and Yang [23] examined the performance of three linear classification methods on text categorization, under 1%, 3%, 5%, 10%, 15%, 20% and 30% class noise ratios. The results showed a dramatic, yet identical, decrease in the classification performances of the three learners after noise ratio exceeded 3%. Specifically the F1-score measures for the three models ranged from 60% to 60% under 5% noise ratio and from 40% to 43% under 30% noise ratio. Pechenizkiy et al. [44] experimented on 8 data-sets the effect of class noise on supervised learning in medical domains. The kNN, Naïve Bayes and C4.5 decision tree learning algorithms were trained on the noisy datasets to evaluate the impact of class noise on accuracy. The classification accuracy for each classifier was compared under eleven class noise levels 0%, 2%, 4%, 6%, 8%, 10%, 12%, 14%, 16%, 18%, and 20%. The results showed that when the level of noise increases, all classifiers trained on noisy training sets suffer from decreasing classification accuracy. Abellan and Masegosa [24] conducted an experiment to compare the performance of Bagging Credal decision trees (BCDT) and Bagging C4.5 in the presence of class noise under 0%,5%,10%,20% and 30% ratios. Both bagging approaches were negatively impacted by class noise, although BCDT was more robust to the presence of noise at a ratio above 20%. The accuracy of BCDT model dropped from 86.9% to 78.7% under a noise level of 30% whereas the Bagging C4.5 accuracy dropped from 87.5% to 77.2% under the same level.

3.3.2 Text Mining for Test Case Selection and Defect Prediction

A multitude of early approaches have used text mining techniques for leveraging early prediction of defects and test verdicts using ML algorithms. However, these studies omit to discuss the effect of class noise on the quality of the learning predictors. In this paper, we highlight the results of some of these work and validate the impact of class noise on the predictive performance of a model for TCS using the method proposed in [77].

A previous work on TCS [77] utilized text mining from source code changes for training various learning classifiers on predicting test case verdicts. The

method uses test execution results for labelling code lines in the relevant tested commits. The maximum precision and recall achieved was 73% and 48% using a tree-based ensemble. Hata et al. [78] used text mining and spam filtering algorithms to classify software modules into either fault-prone or non-fault-prone. To identify faulty modules, the authors used bug reports in bug tracking systems. Using the 'id' of each bug in a given report, the authors tracked files that were reported as defective, and consequently performed a 'diff' command on the same files between a fixed revision and a preceding revision. The evaluation of the model on a set of five open source projects reported a maximum precision and recall values of 40% and 80% respectively. Similarly, Mizuno el al. [85] mined text from the ArgoUML and Eclipse BIRT open source systems, and trained spam filtering algorithms for fault-prone detection using an open source spam filtering software. The results reported precision values of 72-75% and recall values of 70-72%. Kim et al. [79] collected source code changes, change metadata, complexity metrics, and log metrics to train an SVM model on predicting defects on file-level software changes. The identification of buggy commits was performed by mining specific keywords in change log messages. The predictor's quality on 12 open source projects reported an average accuracy of 78% and 60% respectively.

3.4 Experiment Design

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

3.4.1 Data Collection Method

We worked with 82 test execution results (passed or failed) that belonged to 12 test cases and their respective tested code (overall 246,850 lines of code)[1]. First, we used the formula presented in section 3.2 to measure the level of class noise in the data-set - this would help us understand the actual level of class noise found in real-world data-sets. Applying the formula indicated a class noise level of 80.5%, with 198,778 points identified as contradictory. For the remainder of this paper, we will use the term 'code changes data-set' to refer to this data-set. Our first preparation task for this experiment was to convert the code changes data-set into line vectors. In this study, we utilized a bi-gram BoW model provided in an open source measurement tool [43] to carry out the vector transformation. The resulting output was a sparse matrix with a total of 2251 features and 246,850 vectors. To eliminate as many confounding factors as possible, we used the same vector transformation tool and learning model across all experimental trials, and fixed the hyper-parameter configurations in both the vector transformation tool and the learning model (see section 3.5.3)

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

3.4.2 Independent Variable and Experimental Subjects

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

3.4.3 Dependent Variables

The dependent variables are four evaluation measures used for the performance of an ML classifier – Precision, Recall, F1-score, and Matthews Correlation Coefficient (MCC) [86]. The four evaluation measures are defined as follows:

- Precision is the number of correctly predicted tests divided by the total number of predicted tests.
- Recall is the number of correctly predicted tests divided by the total number of tests that should have been positive.
- The F1-score is the harmonic mean of precision and recall.
- The MCC takes the four categories of errors and treats both the true and the predicted classes as two variables. In this context, the metric calculates the correlation coefficient of the actual and predicted test cases for both classes.

3.4.4 Experimental Hypotheses

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

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

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

3.4.5 Data Analysis Methods

The experimental data were analyzed using the scikit learn library with Python [75]. To begin, a normality test was carried out using the Shapiro-Wilk test to decide whether to use a parametric or a non-parametric test for analysis. The results showed that the distribution of the four dependent variables did not deviate significantly from a normal distribution (see section 3.6.2 for details). As such, we decided to use two non-parametric tests, namely: Kruskal-Wallis and Mann-Whitney. To evaluate the hypotheses, the Kruskal-Wallis was selected for comparing the median scores between the four evaluation measures under the treatment levels. The Mann–Whitney U test was selected to carry out a pairwise comparison between the evaluation measures under each treatment level and the same measures at a 0% noise level.

3.5 Experiment Operations

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

3.5.1 Creation of The Control Group

To support the investigation of the hypotheses, a control group was needed to establish a baseline for comparing the evaluation measures under the six treatment levels. This control group needs to have a 0% ratio of class noise, i.e. without contradictory entries. To have control over the noise ratio in the treatment groups, these will then be created by seeding noise into copies of the control group data-set (see Section 3.5.2). The classification performance in the treatment groups will then be compared to that in the control group (see Section 3.5.3). In addition, the distribution of data points in the control group is expected to strongly influence the outcome of the experiment. To control for that we aim to create optimal conditions for the algorithm. ML algorithms can most effectively fit decision boundary hyper-planes when the data entries are similar and linearly separable [87]. Therefore, we decided to start from our industrial code changes data-set (See Section 3.4.1) and extract a subset of the data, by detecting similar vectors in the "Bag of Words" sparse matrix. In this study, we decided to identify similarity between vectors based on their relative orientation to each other. What follows is a detailed description of the algorithm used for constructing the control group. The algorithm starts by loading the feature vectors from our industrial code changes data-set and their corresponding label values (passed or failed) into a data frame object. To establish similarity between two vectors we use the cosine similarity function provided in the scikit learn library [75] working with a threshold of 95%. For each of the two classes (passed or failed), one sample feature vector is randomly picked and used as a baseline vector to compare its orientation against the remaining vectors within its class. The selection criterion of the two baseline vectors is that they are not similar. This is important to guarantee that the derived control group has no contradictory entries (noise ratio = 0). Each of the two baseline vectors is then compared with the remaining vectors (non-baseline) for similarity. The only condition for selecting the vectors is based on their

similarity ratio. If the baseline and the non-baseline vectors are similar more than the predefined ratio of 95%, then the non-baseline vector is added to a data frame object. Table 3.1 shows the two baseline entries before being converted into line vectors. Due to non-disclosure agreement with our industrial partner, words that are not language specific such as variable and class names are replaced with other random names.

Table 3.1: The Two Baseline Entries Before Coversion

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

The script for generating the datasets is found at the link in the footnote[2]. The similarity ratio of 95% was chosen by running the above algorithm a multiple times using five ratios of the predefined similarity ratio. The criterion for selecting the optimal threshold was based on the evaluation measures of a random forest model, trained and tested on the derived control data-set. That is, if the model's Precision and Recall reached 100%, i.e. made neither false positive nor false negative predictions, then we know that control group has reached sufficient similarity for the ML algorithm to work as efficient as possible. The following threshold values of similarity were experimented using the above algorithm: 75%, 80%, 85%, 90%, and 95%. Experimenting on these ratios with a random forest model showed that a ratio of 95% cosine similarity between the baseline vector and the rest yield a 100% of Precision, Recall, F1-score, and MCC. As a result, we used a ratio of 95% to generate the control group. The resulting group contained 9,330 line vectors with zero contradictory entries between the two classes. The distribution of these entries per class was as follow:

- Entries that have at least one duplicate within the same class: 3,679 entries labeled as failed and 4,280 entries as pass.
- Entries with no duplicates in the data-set: 1 entry labeled as failed and 1,370 entries as passed.

3.5.2 Class Noise Generation

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

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

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

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

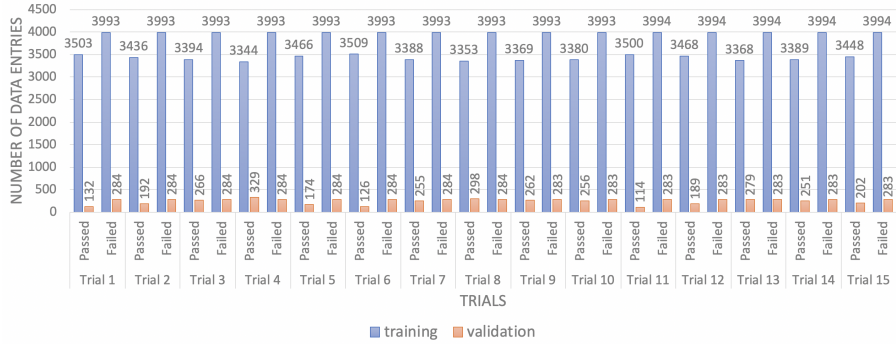


Figure 3.2: The Distribution of The Binary Classes After Generating Noise at 10% Ratio.

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

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

3.5.3 Performance Evaluation Using Random Forest

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

3.6 Results

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

3.6.1 Descriptive Statistics

The descriptive statistics are presented in Tables 4.4, 4.5, 3.4, and 3.5 individually for each dependent variable. The values for Precision (Table 4.4), Recall (Table 4.5), F1-score (Table 3.4), and MCC (Table 3.5) are shown for each of the noise ratio (0%, 10%, 20%, 30%, 40%, 50%, and 60%). A first evident observation from the tables is that there is a statistically significant relationship between the mean values of the four dependent variables and the noise ratio, where a lower value of a given dependent variable indicates higher noise ratio. Three general observations can be made by examining the data shown in the four tables:

- There is an inverse trend between noise ratio and learning precision, F1-score, and MCC. That is, when the noise level increases, the classifier trained on noisy instances suffers a small decrease in the four evaluation measures. Figure 3.3 shows this relationship where the x-axis indicates the noise ratio and the y-axis represents the evaluation measures.
- There exists a higher dispersion in the evaluation scores when the noise level increases (i.e. higher standard deviation [SD]).
- The mean difference between the recall values under each noise ratio is relatively smaller than those with the other three dependent variables.

Table 3.2: Descriptive Stats For Precision.

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

Table 3.3: Descriptive Stats For Recall.

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

3.6.2 Hypotheses Testing

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

Table 3.4: Descriptive Stats For F1-score.

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

Table 3.5: Descriptive Stats For MCC.

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

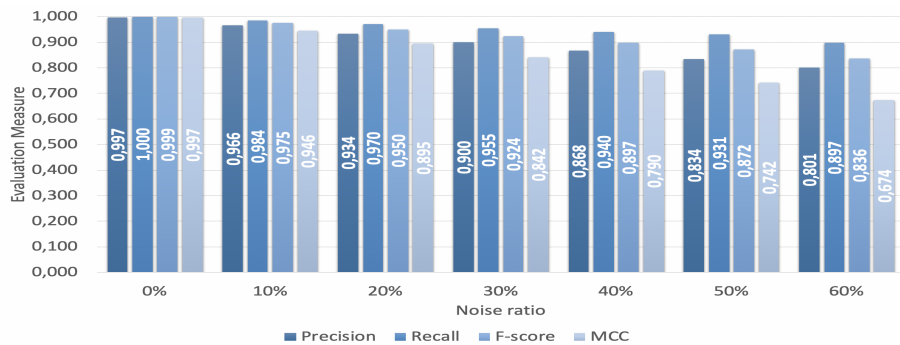


Figure 3.3: Mean Distribution of the Evaluation Measures.

Wilk test results were statistically significant for all the evaluation measures in the majority of the noise ratios. Table 3.6 shows the statistical results of normality for the dependent variables on all noise ratios. These results indicate that the assumption of normality in the majority of the samples can be rejected, as indicated by the p-value ($p < 0.05$) in Table 3.6. Since we have issues with normality in the majority of samples, we decided to run a non-parametric test for comparing the difference between the performance scores under the six noise ratios.

To examine the impact of class noise on the four dependent variables, the Kruskal-Wallis test was conducted. Table 3.7 summarizes the statistical comparison results, indicating a significant difference in Precision, F1-score, and MCC. Specifically, the results of the comparison for precision showed a test statistics of 56.8 and a p-value below 0.001. Likewise, a significant difference in the comparisons between the evaluation measures of F1-score and MCC

(F1-score Results: Test Statistics = 54.172, p-value <0.005, MCC Results: Test Statistics = 53.398, p-value <0.005) groups was found. In contrast, no significant difference between the Recall measures was identified.

Table 3.6: Statistical Results For Normality.

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

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

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

The Mann–Whitney U test with Precision, F1-score, and MCC as the dependent variables and noise ratio as the independent variable revealed a significant difference (p-value below 0.005) under each of the six levels when compared with the same measures in the no-treatment sample. However, the statistical results for recall only showed a significant difference when the noise level exceeded 30%. Table 3.8 summarizes the statistical results from the Mann–Whitney test under the six treatment levels. The analysis results from this experiment indicate that *there is a statistical significant difference in predictive performance for a test case selection model in the presence and absence of class noise*. The results from the Kruskal-Wallis test were in line with the expectations for hypotheses *H0p, H0f, H0mcc*, which confirm that *we can reject the null hypotheses for H0p, H0f, H0mcc*, whereas *no similar conclusion can be drawn for hypothesis H0r*. While no significant difference between the recall values was drawn from the Kruskal-Wallis test, the Mann-Whitney test indicates that there is a significant inverse causality between class noise and recall when noise exceeds 30%. In the domain of TCS, the practical implications can be summarized as follow:

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

Table 3.8: The Comparison Results From Mann-Whitney Test

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

3.7 Threats to Validity

When analyzing the validity of our study, we used the framework recommended by Wohlin et al. [55]. We discuss the threats to validity in four categories: external, internal, construct, and conclusion.

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

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

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

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

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

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

Use of a single ML model. This study employed a random forest model to examine the effect of class noise on classification performances. However, the analysis results might differ when other learning models are used. This was a

design choice since we wanted to study the effect of a single treatment and to control as many confounding factors as possible.

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

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

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

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

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

3.8 Conclusion and Future Work

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

There are still several questions that need to be answered before concluding that class noise handling strategies can be used in an industrial setting. A first question is about finding the best method to handle class noise with respect to efficiency and effectiveness. Future research that study the impact of attribute noise on the learning of a classifier and how that compares with the impact of

class noise are needed. Other directions for future research include evaluating the level of class noise at which ML can be deemed useful by companies in predicting test case failures, evaluate the relative drop of performance from a random sample of industrial code changes and compare the performance of the learner with the observations drawn from this experiment, study and compare the effect of different code formatting on capturing noisy instances in the data and the performance of a classifier for TCS. Finally, we aim at comparatively exploring the sensitivity of other learning models to class and attribute noise.

Chapter 4

Paper C

Improving Test Case Selection By Handling Class and Attribute Noise

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

Accepted for publication in Journal of Systems and Software.

Abstract

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

4.1 Introduction

Machine Learning (ML) models have been increasingly used for automating software engineering activities [79, 88–90]. One example for the use of ML models is optimizing software regression testing in continuous integration (CI), where ML is used to recommend which test cases should be included in test suites to reduce the cost overhead for testing resources. Since regression testing is performed frequently (after every commit), they result in large quantities of data that include test execution results. This poses an opportunity to utilize ML when such large data is available for analyses.

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

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

Several approaches in the literature sought to address the problem of defects prediction and test case selection in CI. Examples include static code analysis [91], [92], static code metrics [73], [58], natural language processing (NLP) [77], [93]. These approaches use data-sets with historical defects for

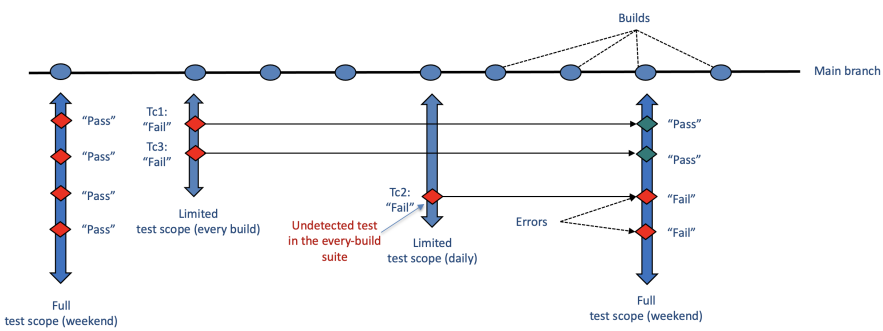


Figure 4.1: CI Pipeline with Test Case Selection.

training machine learning (ML) models to classify code as either non-defective

or defective (i.e. in need for testing) or to predict whether test cases will fail. In our previous work [77], we studied an industrial case of the use of ML classifiers and textual analysis to predict test case execution results. The method was evaluated on a data-set whose size was 1.4m lines of code (LOC).

However, one of the challenges in building a learner for predicting test case execution results lies in the amount of noise that comes in the data. This challenge is particularly important in the domain of testing, since frequent automated executions of test cases can introduce noise in an uncontrolled way. A complete taxonomy of noise types is still an open research issue [38]. However, two categories of noise types are most commonly addressed in the literature - class and attribute noise [17, 37, 84, 94]. Class noise (also known as label or annotation noise) occurs as a result of either contradictory entries or mislabeling training entries [84], whereas attribute noise occurs due to either selecting attributes that are irrelevant for characterizing the training instances and their relationship with the target class, or using redundant or empty attribute values [84], [95].

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

A number of research studies proposed several techniques for handling class and attribute noise [17], [18], [19], [20]. Those can be classified into three broad categories: tolerance, elimination/filtering, and correction/polishing. In the tolerance category, imperfections in the data are dealt with by leaving the noise in place and designing ML algorithms that can tolerate a certain amount of noise. Approaches in the elimination category seek to identify noisy entries and remove them from the data set. Entries that are suspected to be spurious (e.g., mislabelled or redundant) are discarded and removed from the training data. In the last category, instead of removing the corrupted entries, those entries get repaired by replacing their values with more appropriate ones. There are a number of advantages and disadvantages associated with each one of these approaches. In the tolerance category, no time needs to be invested on cleaning the data, but a learner built from uncleaned data might be less effective. By filtering noisy instances, we compromise information loss in the interest of retaining cleaner instances of the data. By carrying out correction of noisy instances, we introduce risks of presenting undesirable attributes but preserve maximal information in the data.

In a previous work [93], we introduced an approach for addressing the problem of annotation noise by relabeling contradictory entries and removing duplicate ones in one of the classes. The empirical evaluation of applying the technique was measured with respect to precision, recall, and F1-score using an industrial data. In this study, we extend that work by examining the effect

of applying an attribute noise elimination approach, called Pairwise Attribute Noise Detection Algorithm (PANDA) [25], on the performance of a TCS learner using the class-noise cleaned data reported in [93]. The purpose is to provide testers with insights into choosing the right noise handling strategies and to counteract exhaustive efforts on noise cleaning for more effective TCS. For this, we design and implement a controlled experiment on the same industrial data-set used in our previous study [93] and examine the effect of removing training observations that come with high attribute noise on learning. Specifically, we address the following research question:

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

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

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

4.2 Related Work

Many research efforts have been made to handle class and attribute noise for improving the predictive quality of learners. However, studies that investigate the impact of class and attribute noise handling in the domain of software engineering is lacking [15]. In this section, we begin by highlighting work that leverage the use of ML models for early prediction of defects and test case verdicts for test selection. Thereafter, we highlight related work that examine the effect of class and attribute noise on learning performances.

4.2.1 Text Mining For Defect Prediction and Test Case Selection

A multitude of early approaches have used text mining techniques for leveraging early prediction of defects and test case verdicts using various learning algorithms and statistical approaches. However, these studies omit to discuss the effect of class noise on the quality of the learning predictors. As a result, in

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

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

this paper, we mention some of these previous work, as summarized in Table 4.1

A previous work on test case selection [77] utilized text mining from source code changes for training various learning classifiers on predicting test case verdicts. The method uses test execution results for labelling code lines in the relevant tested commits. The maximum precision and recall achieved was 73 and 48 percent using a tree-based ensemble. Hata et al. [78] used text mining and spam filtering algorithms to classify software modules into either fault-prone or non-fault-prone. To identify faulty modules, the authors used bug reports in bug tracking systems. Using the 'id' of each bug in a given report, the authors tracked files that were reported as defective, and consequently performed a 'diff' command on the same files between a fixed revision and a preceding revision. The evaluation of the model on a set of five open source projects reported a maximum precision and recall values of 40 and 80 percent respectively.

Similarly, in an earlier work, Mizuno el al. [85] mined text from the ArgoUML and Eclipse BIRT open source systems, and trained spam filtering algorithms for fault-prone detection using an open source spam filtering software. The

results reported a precision of 72-75 percent and a recall of 70-72.

Aversano et al. [80] extracted a sequence of source code snapshots from two version control systems and trained five learning algorithm to predict whether new code changes are defective or not. The K-Nearest Neighbor predictor performed better than the other algorithms with a good trade-off between precision and recall, yielding precision and recall values of 59-69 percent and 59-23 percent respectively.

Kim et al. [79] collected source code changes, change metadata, complexity metrics, and log metrics to train an SVM model on predicting defects on file-level software changes. The identification of buggy commits was performed by mining specific keywords such as 'Fixed' or 'Bug' in change log messages. Once a keyword is found, the assumption that changes in the associated commit comprise a bug fix is made, and hence used for labelling code instances in the relevant commit. The predictor's quality on 12 open source projects reported an average accuracy of 78 and 60 percents respectively.

4.2.2 Class Noise Handling Research

Brodley et al. [18] uses an ensemble of classifiers, named Consensus Filter (CF), to identify and remove mislabeled instances. Using a majority voting mechanism with the support of several supervised learning algorithms, noisy instances are identified and removed from the training set. If the majority of the learning algorithms fail to correctly classify an instance, a tag is given to label the misclassified instance as noisy and later tossed out from analysis. The evaluation results show that when the class noise level is below 40%, filtering results in better predictive accuracy than not filtering. On the basis of their experiments, the authors suggest that using any types of filtering strategies would improve the classification accuracy more than not filtering.

Al-Sabbagh et al. [96] conducted a controlled experiment to examine the effect of class noise at six levels on the learning performance for a test selection model. The analysis was done on an industrial data for a software program written in the C++ language. The results revealed a statistically significant relationship between class noise and the precision, F1-score, and Mathew Correlation Coefficient under all the six noise levels. Conversely, no similar relationship was found between recall and class noise under 30% noise level. The conclusion drawn suggested that higher class noise ratio leads to missing out more tests in the predicted subset of test suite. Moreover, it increases the rate of false alarms when the class noise ratio exceeds 30%.

Guan et al. [17] introduced CFAUD, a variant of the approach proposed by Brodley et al. [18], which involves a semi-supervised classification step in the original approach to predict unlabeled instances. The approach was tested for an effect on learning for three ML algorithms (1-NN, Naive Bayes, and Decision Tree) using benchmark data-sets. The empirical results indicate that both majority voting and CFAUD have a positive effect on the learning of the three ML algorithms under four noise levels (10%, 20%, 30%, and 40%). However, averaged on the four noise levels, the improvement that CFAUD provides over CF is around 12% for each of the three classifiers.

Muhlenbach et al. [37] introduced an outlier detection approach that uses neighbourhood graphs and cut edge weight algorithms to identify mislabeled

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

4.2.3 Attribute Noise Handling Research

Khoshgoftaar et al. [38] presented a rule-based approach that detects noisy observations using Boolean rules. Observations that are detected as noisy are removed from the data before training. The approach was compared for efficiency and effectiveness against the C4.5 consensus filter algorithm presented in [18]. The results drawn from the case study suggests that when seeding noise in 1 to 11 attributes at two noise levels, the consensus filter outperforms the rule-based approach. Conversely, the rule-based approach outperforms the other approach with respect to efficiency.

Khoshgoftaar et al. [19] proposed an approach that computes noise ranks of observations relative to a user defined attribute of interest. A case study for evaluating the approach was conducted on data derived from a software project written in C and consists of 10,883 modules. In their study, the attribute of interest was chosen to be the class attribute. A comparison between the efficiency and effectiveness of the method in detecting noise and a popular classification filter algorithm [18] was made. The results reported different effectiveness scores ranging from 24% to 100% effectiveness.

Khoshgoftaar et al. [97] extended their work in [25] and proposed an approach that identifies noisy attributes in the data. Upon identifying attributes that are least correlated with the target class, those attributes get eliminated from the analysis. The approach is based on the Kendall's Tau rank correlation to identify weakly correlated attributes with the target attribute. In terms of evaluation, the effectiveness of the technique was studied using two data-sets belonging to assurance software projects, where an inspection of a software engineering expert was done to judge the performance of the approach. The overall results suggest a strong match between the output of the approach and the observations drawn from an expert in the field.

Teng [98] studied the effectiveness of three noise handling approaches, namely robustness, filtering, and correction using decision trees built by C4.5. Twelve machine learning data sets were used for the evaluation. The classification accuracy of the learners suggest that elimination and correction are viable mechanisms for minimizing the negative impact of noise. In particular, using an elimination approach before building a classifier reported an accuracy score that ranged from 77% to 100%.

Quinlan [99] demonstrated that as the noise level in the data increases, removing attribute noise information from the data decreases the predictive performance of inductive learners if the same attribute noise is present in

other attributes in the data to be classified. Similarly, Zhu and Wu concluded, following a number of experiments, that attribute noise is not as harmful as class noise on the predictive performance of ML models [84].

While the majority of these work emphasize on the importance of handling both class and attribute noise in data for improving the predictive performance, the results from our study provide counter-evidence that opposes these findings when it comes to attribute noise. More precisely, the analysis results demonstrate that removing training observations that come with high attribute noise has no effect on the predictive performance of an ML classifier. These results are in line with the findings drawn by Quinlan, Brodley and Friedl, and Zhu and Wu [99], [100], [84].

4.3 Background, Definitions, and Examples

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

4.3.1 Core Concepts

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

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

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

MeBoTS is a machine learning based method that functions as a predictor of test case verdicts [77]. The method employs a predictive model that learns

from historical code churns and their relevant test case verdicts for predicting lines that would trigger test case failures. The method is described in 3 steps.

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

Textual Analysis and Features Extraction (Step 2) The second step in the method involves transforming the collected code changes into feature vectors. For this purpose we used an open source tool [43] that utilizes the Bag of Words (BoW) approach for modelling textual input. The tool uses each line from the extracted code churns in step 1 and:

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

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

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

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

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

Method	**Description**	**Pros and Cons**	**MeBoTs**
Code metrics	Uses code static metrics, such as code complexity, size, churn metrics to train machine learning models on classifying defective code. Examples: [101], [102]	**Pros:** - Strong empirical evidence that supports the use of some code metrics for defects prediction for Java programs. **Cons:** - Static metrics need to be decided a priori, and they depend on the size.	- language agnostic and can be applied on any programming language. - The features from MeBoTS are not decided a priori, and are not dependent on size
Static Code Analysis	Uses machine learning models to learn semantic features derived from abstract syntax trees. Examples: [91], [92], [103]	**Pros:** - Characterize defects using abstract syntax tree information from the code. **Cons:** - Code needs to be compiled. - Does not scale well when the number of tree nodes increases.	- Generates feature vectors from the actual program using textual analysis -Does not require the code to be compiled. - Uses statistics to generate its feature vectors.
Dynamic Analysis	This category relies on executing the program and comparing its actual with expected behavior. Examples: [104], [105]	**Pros:** -Allows for analysis of the program without having access to the code. **Cons:** - If the code does not run, no analysis is done	- Analyzes the code before compiling the program.

4.3.3 Noise Definitions and Examples

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

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

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

4.3.3.2 Definition and Example of Class Noise in Code Churns Data

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

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

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

As an illustration of the class noise problem in a data-set consisting of code churns, Figure 4.3 shows a sample C++ program transformed into feature

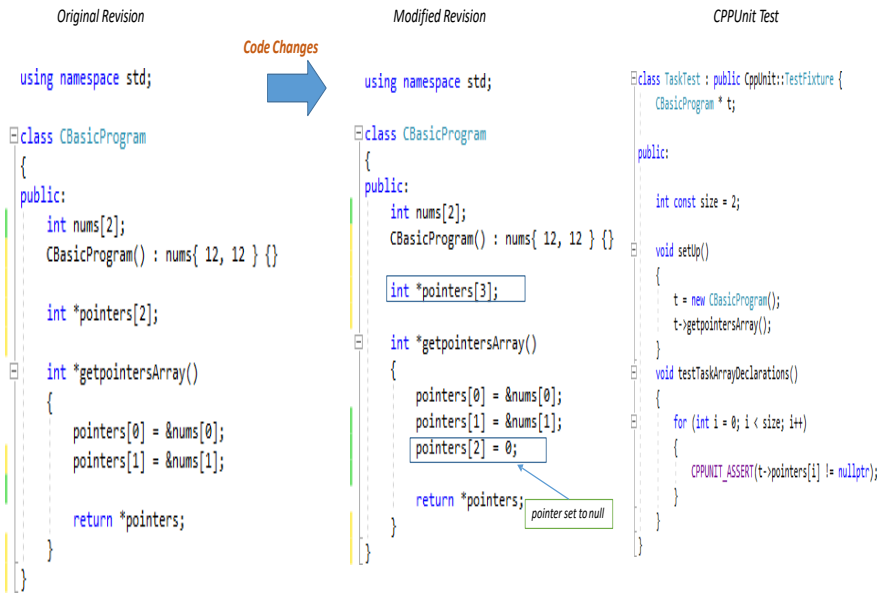


Figure 4.2: Example On the Relationship Between Code Churns and Test Case Verdicts

vectors using the BoW approach. Each line of code in the sample program is transformed into a line vector which gets assigned a class value based on a *tce* result for the committed code. These transformed lines and their relevant *tce* get fed as input into an ML model for training. The model is used to predict which lines in the program will trigger a test case failure or success.

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

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

4.3.4 Definition and Example of Attribute Noise in Code Churns Data

The definition of attribute noise in this paper follows the one proposed by Van Hulse et al. [25], which suggests that a noisy observation appears when one or more of its attributes deviates from the general distribution of other attributes. The larger the deviation is for one or more observations, the more evidence there is that they are noisy. In the context of the given problem (i.e., TCS),

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

Figure 4.3: Class Noise Example in Code Base.

a deviation between attributes can occur when the general distribution of S follows a standard coding style, whereas a smaller fraction of S deviates from the standard.

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

4.4 Noise Handling and Removal Approaches

The problem of achieving a good learning performance in the presence of noisy environments has been widely highlighted in the ML literature. Several approaches have been built to enhance the learning performance of ML classifiers

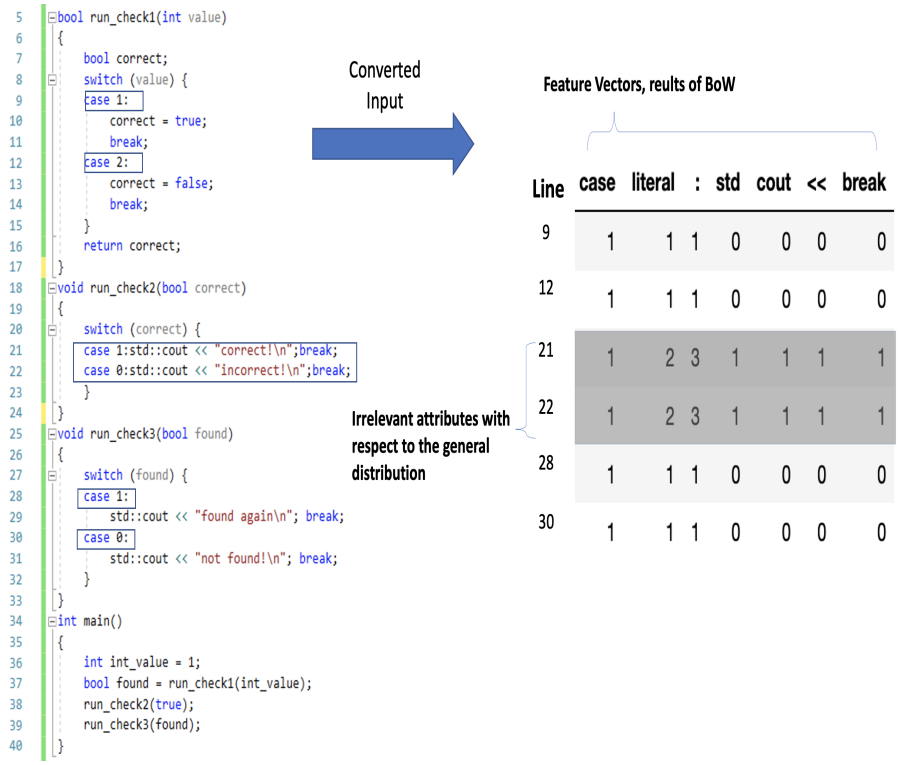


Figure 4.4: Attribute Noise Example in Code Base.

[83], [84], [37]. Nevertheless, the presence of class and attribute noise have been reported to still have a negative influence on the learning, and thus needs to be handled before training. In this section, we describe an approach that we introduced in the baseline study [93] to handle the problem of class noise. Thereafter, an existing elimination based approach from the literature for handling attribute noise is described.

4.4.1 Class Noise Approach

Our approach for handling annotation noise relies on relabeling repeated code lines that come with different class values. These repeated lines can potentially occur in code churns due to several scenarios, such as 1) copying of code [106], and 2) merge transactions [94]. The first scenario manifests itself in the event of 'copy-paste' reuse of code check-ins that had previously passed the testing and integration phases. In such scenario, the developers explicitly duplicate source code fragments to adapt the duplicates for a new purpose in a quick fashion [106]. The second scenario appears when developers in one or more teams work on dedicated branches for features development and use similar code phrases as to those committed and merged from different branches [94] e.g., `x = x + 1;`. When extracting such code check-ins with duplicate code phrases for TCS, inconsistent observations with different class values might occur.

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

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

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

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

4.4.2 Selected Attribute Noise Handling Approach

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

To address the problem of attribute noise in training data, we decided to use an existing elimination based approach called PANDA [25] that identifies training instances with large deviations from normal. The PANDA algorithm identifies such instances by comparing pairs of attributes in the space of feature vectors. The output is an ordered list of noise scores for each code line - the higher the noise score for a code line, the higher it deviates from normal. Upon ranking noisy instances, the generated list can be used to toss out instances (code lines) that come with the highest rank with respect to attribute noise.

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

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

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

4.5 Research Methodology

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

4.5.1 Original Data Set

In the baseline paper [93], we worked with a data set of code churns that belong to a legacy system written in the C language. A total of 82 test case execution results (35 passed tests and 47 failed tests) for 12 test cases and their relevant set of code changes (1.4 million LOC) were collected. The system from which the sample data was extracted belongs to a large Swedish telecommunication company and has the size of several million lines of code. The feature vectors generated from the data-set in [93] using a bi-gram BoW model comprised a total of 2251 features/attributes. The distribution of the binary classes in the collected data was fairly balanced, with 44% of the code lines belonging to the 'passed' class and 56% to the 'failed' class [3].

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

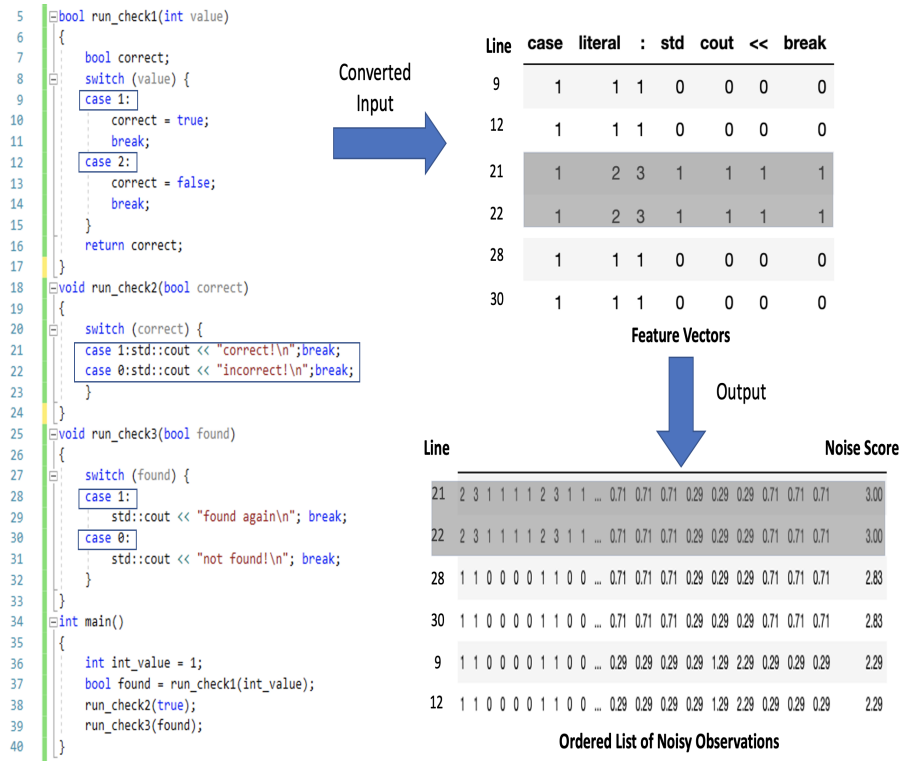


Figure 4.5: An Excerpt of PANDA's Output

4.5.2 Random Forest For Evaluation

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

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

4.5.3 Class Noise

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

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

4.5.4 Attribute Noise

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

4.5.4.1 Adopted Data-Set

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

According to Ganganwar and Vaishali [107], a data-set is called imbalanced when it contains many more samples under one class than from the rest of the classes. Accordingly, we inspected the distribution of the samples in the control group with respect to the binary classes (defective and non-defective) in order to determine the balance of classes. Figure 4.6 shows that the distribution of instances in the non-defective class contains many more samples than the defective class (14,400 to 5,415 samples). Based on this distribution and given that we only have two classes (binomial distribution), we consider the control group to be imbalanced. To overcome this problem, we chose to upsample instances in the minority class using the 'resample' module provided in the Scikit-learn library [75]. The idea of oversampling is to randomly generate samples from the minority class instances until the number of minority class is the same as the number of majority class.

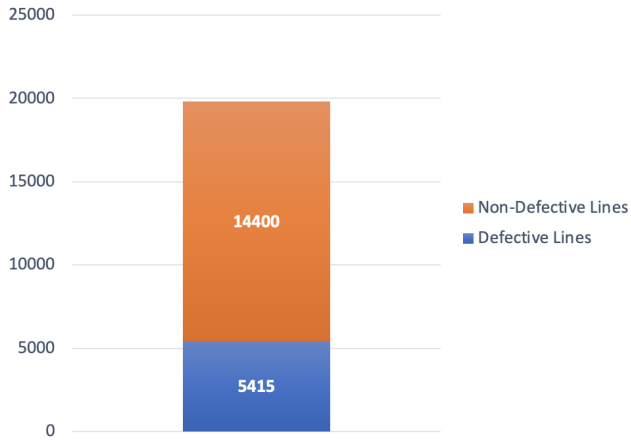


Figure 4.6: The Distribution of Classes In The Adopted Data-Set

4.5.4.2 Independent Variable and Experimental Subjects

In this study, attribute noise removal was the only independent variable (treatment) examined for an effect on classification performance. Ten variations of the treatment were selected. Namely, 5%, 10%, 15%, 20%, 25%, 30%, 35%, 40%, 45%, 50%. Each treatment level corresponds to a fraction size of observations that gets eliminated before training the ML model in MeBoTS. We used 25-fold stratified cross validation on the control group to derive 25 experimental subjects on which the treatment is applied. Each subject is treated as a hold out group for validating an RF model which gets trained on the remaining 24 training subjects. A total of 275 trials derived from the 25-folds were conducted - each 25 trials for evaluating the performances of a learner under one treatment level.

4.5.4.3 Dependent Variables

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

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

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

4.5.4.4 Experimental Hypotheses

Three hypotheses are defined according to the goals of this study and tested for statistical significance in Section4.5.4.5. The hypotheses were based on

the assumption that data-sets with more attribute noise have a significantly negative impact on the classification performance of an ML model for TCS compared to data with no attribute noise. The hypotheses are as follow:

- *H0p: The mean Precision is the same for a model with and without attribute noise*

$$\mu 1p = \mu 2p \tag{4.1}$$

.

- *H0r: The mean Recall is the same for a model with and without attribute noise*

$$\mu 1r = \mu 2r \tag{4.2}$$

.

- *H0f: The mean F1-score is the same for a model with and without attribute noise*

$$\mu 1f = \mu 2f \tag{4.3}$$

.

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

4.5.4.5 Data Analysis Methods

The experimental data were analyzed using the scikit learn library [75]. To decide whether to use a parametric or non-parametric test for the analysis, a normality test was carried out. First, we plotted the frequency distribution graphs for the three dependent variables under each treatment level to see if they deviate from a normal distribution. To further validate the visual inspection, a Shapiro-Wilk test was carried out. The results showed that 3 dependent variables are not normally distributed (see Section 4.6.2 for details). Based on the normality test results, we decided to use two non-parametric tests, namely: Kruskal-Wallis and Mann-Whitney. To evaluate the hypotheses, the Kruskal-Wallis was selected for comparing the median scores between the three evaluation measures under the 11 treatment levels. The Mann–Whitney U test was selected to perform a pairwise comparison between the evaluation measures under each treatment level and the same measures with no treatment (0% noise removal).

4.5.4.6 Attribute Noise Removal

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

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

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

Table 4.3: An Excerpt Of the Output Generated From PANDA

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

4.6 Evaluation Results

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

4.6.1 Original vs. Class Noise Cleaned Data

The performance measurements of the RF classifier built on the class-noise cleaned data is plotted using a confusion matrix, as shown in Figure 4.7. The Figure shows a non-normalized matrix for the predicted and actual values of test case verdicts for all lines in the test set. The first cell on the upper left hand side corresponds to the number of lines (6,543) that are predicted to trigger test case failures and are actually true. On the same diagonal, the last

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

cell to the bottom right of the matrix indicates the number of lines (15,688) that are predicted to be non-defective and are actually true, and require no testing. The remaining entries in the test set correspond to the number of misclassified lines.

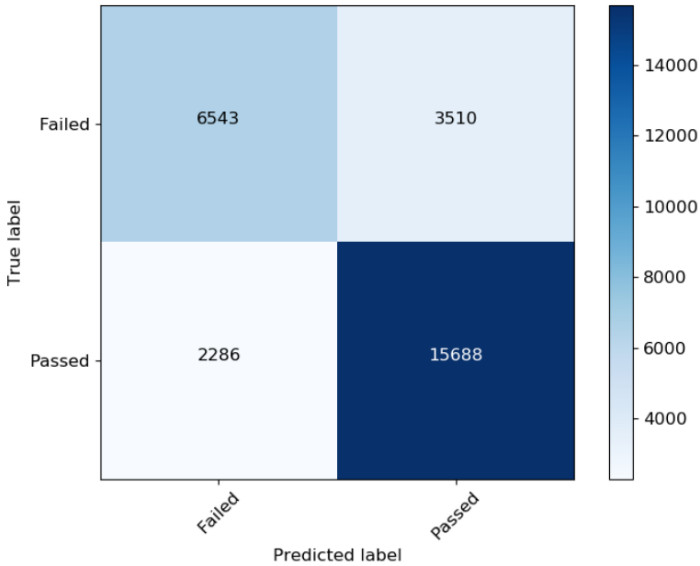


Figure 4.7: Confusion Matrix For a Classifier Trained on Class Noise cleaned Data

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

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

This subsection discusses the results of the descriptive statistics and statistical tests conducted to evaluate hypotheses *H0p, H0r, and H0f* presented in Section 4.5.4.4. The results reported in this section are used for drawing a comparison between the effectiveness of handling class noise and attribute noise on the learning performance. Figures 4.9, 4.10, and 4.11 show three box-plot graphs, which were plotted to visually inspect the effect of removing observations with attribute noise at each treatment level on the dependent variables. A first observation from the graphs suggests a lack of causality between the treatment and the three dependent variables. This observation was further supported by examining the mean scores of each dependent variable in the descriptive statistics, as shown in Tables 4.4, 4.5, and 4.6. Note that the precision, recall, and F1-score reported in the three tables under 0% treatment level are different

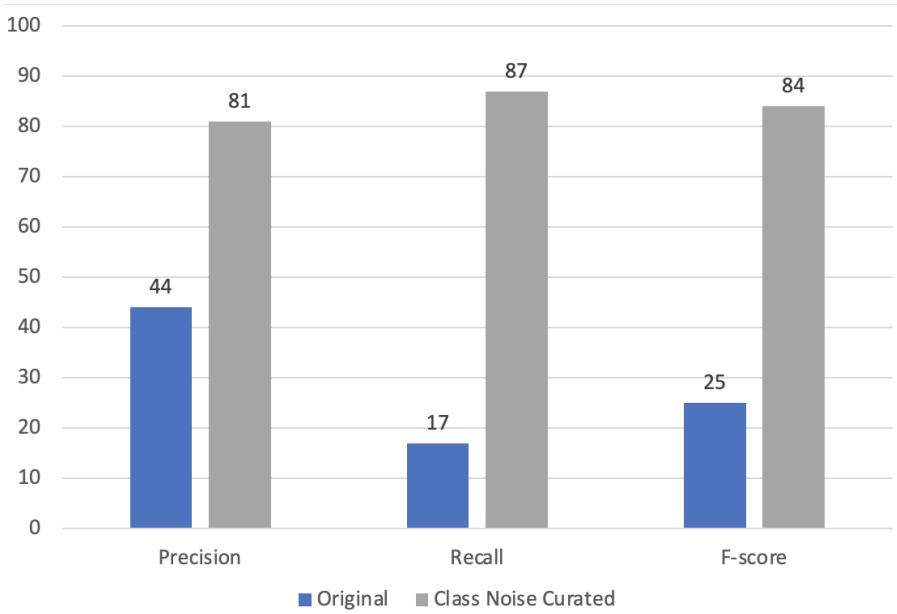


Figure 4.8: Learning Performance On the Original and the Class Noise cleaned Data Sets

than those obtained from training on the class-noise cleaned data. This is because the control group was used as a baseline for the class-noise cleaned data from which the ML model was built.

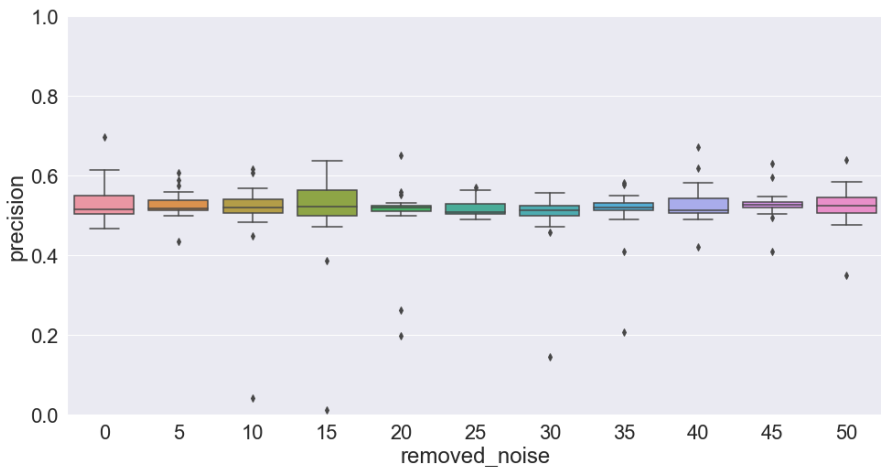


Figure 4.9: The Distribution of Precision Values Under the Treatment Levels

To begin the evaluation of the hypotheses, we start by checking the normality in the distribution of the three dependent variables. The frequency distribution of the variables were plotted for the 275 trials (25 trials for each treatment level) to visually inspect normality, as shown in Figures 4.12, 4.13, and 4.14. Then,

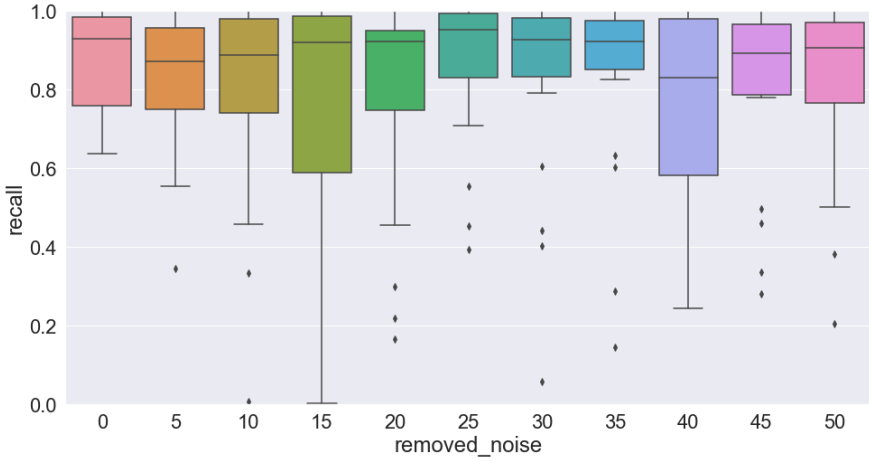


Figure 4.10: The Distribution of Recall Values Under the Treatment Levels

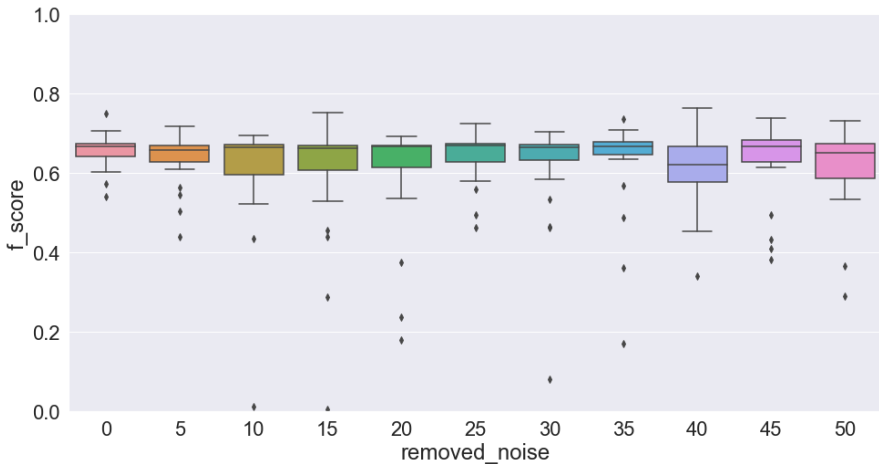


Figure 4.11: The Distribution of F1-score Values Under the Treatment Levels

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

To examine the effect of removing observations with top rank attribute noise on the learning, the Kruskal-Wallis test was conducted. Table 4.9 summarizes the statistical comparison results, indicating no significant difference in Precision, Recall, and F1-score. Specifically, the results of the comparison

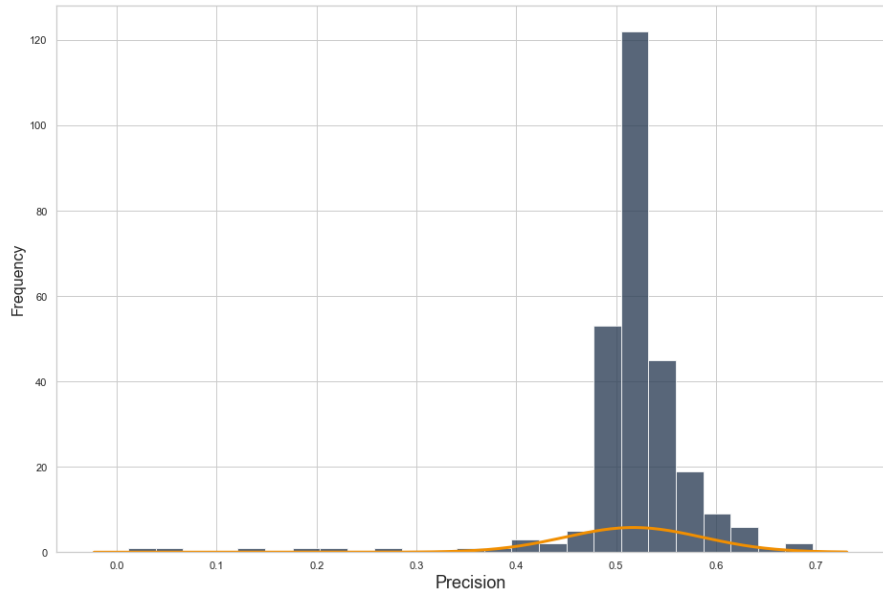


Figure 4.12: Frequency Plot For the Precision Scores

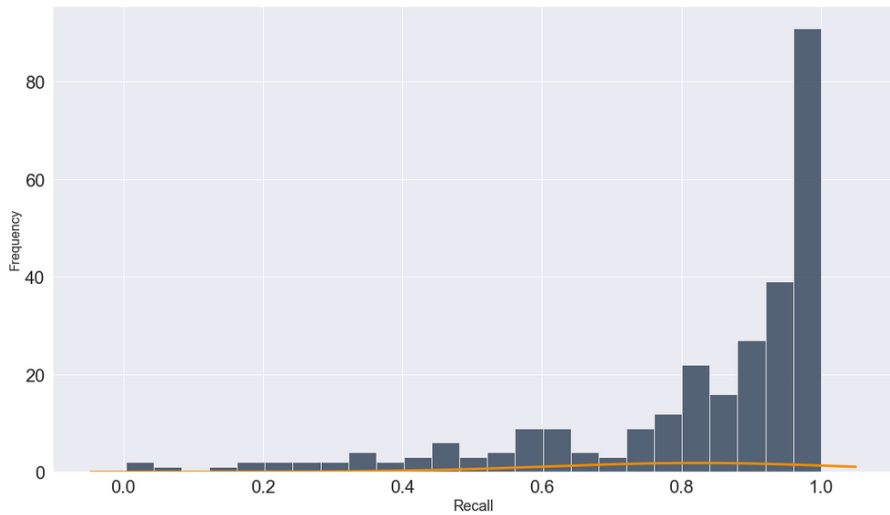


Figure 4.13: Frequency Plot For the Recall Scores

Table 4.4: Descriptive Statistics For Precision.

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

Table 4.5: Descriptive Statistics For Recall.

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

for precision showed a test statistics of 7.96 and a p-value of 0.63. Likewise, a significant difference in the comparisons between the evaluation measures of Recall and F1-score (Recall Results: Test Statistics = 8.62, p-value = 0.56 , F1-score Results: Test Statistics = 8.56, p-value = 0.57) values were not found. Therefore, no statistical evidence could be found to support the rejection of the null hypotheses *H0p, H0r, H0f.*

4.7 Discussion

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

Table 4.6: Descriptive Statistics For F1-score.

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

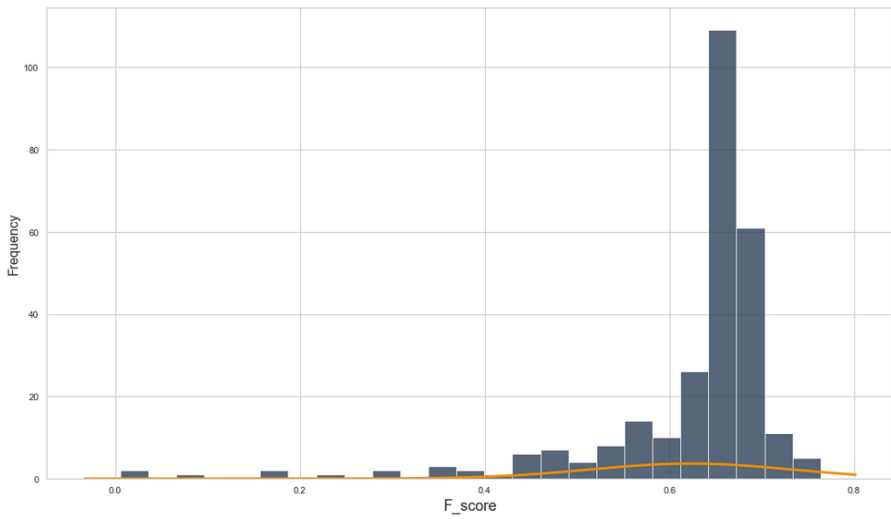


Figure 4.14: Frequency Plot For the F1-score Scores

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

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

attribute noise at different levels on the performance of the ML model used in MeBoTS.

By examining the performance measures in the Tables and Figure, the

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

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

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

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

following observations are drawn from the comparison:

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

These observations imply that training a classifier on a class-noise cleaned data will yield to a better performance with respect to precision and recall than the other two Trials of training. Particularly, the results suggest that building a learner on class-noise cleaned data will allow testers to correctly exclude 8 out of 10 actually passing test cases from execution (81% precision). In addition, the results reveal that training a learner on a PANDA cleaned data would result in building a learner that is biased towards the positive class. The implication that these results bring in the domain of TCS are that tester would falsely exclude 5 out of every 10 actually passing test cases from execution. These results are in line with the conclusions drawn by Brodley and Friedl, and Zhu and Wu [100] [84], which suggest that attribute noise is less harmful than class noise on the inductive performance.

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

- To avoid randomness in the prediction of test case verdicts, uncleaned data should not be used for building a learner for TCS.

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

4.8 Threats to Validity

When analyzing the threats to validity of our study, we followed the framework recommended by Wohlin et al. [55] and discuss the validity in terms of: external, internal, construct, and conclusion.

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

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

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

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

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

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

Configuration. In this study, the ranking of noisy observations produced by PANDA was determined using a bin size of five. Since the binning size in PANDA may affect the ranking of noisy observations [25], there is a likelihood that we chose a bin size that does not identify the highest noisy observations in

the sample data. As a result, the applied treatment may not have eliminated all observations that come with the highest attribute noise. This may have an effect on the learning. However, our results showed that the standard deviations in the learning scores were not largely despaired across the 25 subjects, which means that the effect of the chosen bin size had a similar effect on learning across all experimental subjects.

Instrumentation. A potential internal threat is the presence of undetected issues in the scripts used for vector transformation, data-collection, and PANDA's implementation. This threat was controlled by carrying out a careful inspection of the scripts and testing them on small subsets.

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

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

The Binning Algorithm. The binning algorithm used in the original work of PANDA was not explicitly stated in the original publication [25]. As a result, we used the sort_values function in the PANDA module of the scikit learn library to discretize attribute values into bins of predefined sizes. Thus, our implementation of the algorithm may differ than the one used in the original work. However, the authors of the original publication state that any binning algorithm can be used without affecting the performance.

The Calculation of Noise Score. The description for calculating the standardized noise score in the original publication of PANDA [25] created a confusion with respect to whether the mean and standard deviation should be calculated for each partition in x_j or x_k. On the one hand, the description states that *the standardized noise score for attribute value x_{ik} is calculated relative to the partitioned attribute value for instance i,$\hat{x}_{ik}$*. On the other hand, the description states that *'the mean and standard deviation of the non-partitioned attributes $x_{k,k\neq j}$ relative to each bin $\hat{x}_{j=0,\ldots,L'}$ is calculated'*. In our implementation, we interpreted the relativeness between an attribute value x_{i_k} with the partitioned attribute value for instance i, $\hat{x}_ik$ by subtracting the attribute value x_{ik} from the mean to standard deviation ratio of the bin in x_j relative to x_{ik}. The alternative interpretation would be to subtract x_{ik} from the mean to standard deviation ratio of the elements in x_k relative to the bin in x_j. Nevertheless, our implementation was manually inspected on a small set of line vectors (as shown in Section 4.3.4) and the ranking of noisy observations were in line with the definition of attribute noise provided in the original publication [25].

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

learner was not biased towards a particular class.

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

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

4.9 Conclusion and Future Work

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

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

4.10 Appendix A

Attribute Noise	Performance metrics	Random Forest n_estimater=100	Random Forest n_estimater=300
0%	Acc	0.54	0.53
	Prec	0.53	0.50
	Rec	0.88	0.82
	F-score	0.66	0.62
	MCC	0.13	0.10
5%	Acc	0.54	0.53
	Prec	0.53	0.52
	Rec	0.83	0.84
	F-score	0.64	0.64
	MCC	0.1	0.10
10%	Acc	0.53	0.52
	Prec	0.51	0.51
	Rec	0.80	0.87
	F-score	0.61	0.64
	MCC	0.09	0.09
15%	Acc	0.53	0.52
	Prec	0.51	0.51
	Rec	0.78	0.93
	F-score	0.60	0.66
	MCC	0.08	0.10
20%	Acc	0.52	0.52
	Prec	0.50	0.51
	Rec	0.80	0.95
	F-score	0.61	0.66
	MCC	0.07	0.1

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

Chapter 5

Paper D

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

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

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

Abstract

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

5.1 Introduction

Software testing has evolved to successfully accommodate for the growing demand of higher product quality and faster delivery of releases [108]. Nevertheless, testing has been notoriously costly for its massive resource consumption - accounting for more than 50% of the development life cycle. Therefore, optimizing testing processes becomes pivotal for companies of all sizes to reduce the cost overhead and increase the velocity of software development.

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

To address this problem of regression testing, a number of test case selection approaches have been proposed in the literature [3], [77], and [109]. These approaches seek to improve the effectiveness of test case selection by inferring statistical models that can potentially predict affected test cases given changes in the code base. However, a mutual drawback among these approaches is that they omit to take into account the dependencies between specific types of code changes (e.g., memory and algorithmic changes) and test case types (e.g., performance and security tests) when training predictive models. For example, Al-Sabbagh et al. [77] proposed building a machine learning (ML) model for test selection by mapping history executions of test cases and their relevant code changes without considering what types of test cases are sensitive to the changes in the source code. Similarly, Knauss et al. [3] proposed an automatic recommender that analyzes the frequency in which test cases fail on a particular day given code changes made to software modules irrespective of the types of changes made in the code and their dependencies with specific test case types.

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

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

To address this research question, we constructed a taxonomy, linking the test case types and the categories of source code that can trigger these test cases. First, we began the taxonomy building by identifying and extracting data from the literature to find the test types and categories of code changes and to identify potential synergies between them. Then, we surveyed testers from software companies to construct and design the faceted taxonomy [110] Finally, for two categories, where the survey results were inconclusive, we conducted a workshop with the testers to find the strength of these dependencies.

5.2 Related Work

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

5.2.1 Defect Taxonomies

A widely applicable taxonomy in the software testing literature is Orthogonal Defect Classification (ODC), which was designed by Chillarege et al [111]. The ODC taxonomy defines attributes for the classification of failures. Its main purpose was to identify the root cause of defects and to provide quick feedback to developers about defects' cause in the software process. The ODC can also be used for early detection of faults in static analysis. Several defect taxonomies have been built on the ODC as a starting point to develop different domain-specific taxonomies. For example, Li et al. [112] presented an extended taxonomy of ODC and named it Orthogonal Defect taxonomy for Black-box Defects (ODC-BD). The taxonomy was designed by the motive of increasing testing efficiency and improving the analysis of black-box defects. Evaluated on the analysis of 1860 black-box defects that belong to 40 software projects, the results showed that using ODC-BD reduced the testing effort by 15% in one month compared to the testing efficiency when not using the ODC-BD. Another work conducted by Li et al. [113] adopted ODC to classify web errors for an improved reliability. Their taxonomy classified web errors according to their response code, file type, referrer type, agent type, and observation time.

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

5.2.2 Taxonomies in Software Testing

Software testing has often been confronted with the challenge of unveiling software defects under sever time pressure and limited hardware resources. Due to its importance and practical relevance, several software testing taxonomies have been proposed in the literature. In a systematic literature review study [114], Britto identified a number of studies that present taxonomies in the area of software testing. The majority of these taxonomies, however, provides a classification of the suitability of testing techniques in different contexts. For example, Novak et al. [115] developed a tree-based classification of features that are attributed to existing static code analysis tools. The taxonomy offers a classification of existing static analyzers based on the technology, availability of rules, and the programming languages that each tool supports. Similarly, Vegas et al. [116] classified a set of unit testing techniques and mapped their characteristics with project characteristics to aid the selection of suitable testing approaches based on the project's characteristics. The presented taxonomy comprised a number of criteria such as when to use the testing approach, who to use it, and where it can be used. Felderer et al. [117] presented a classification for supporting the categorization of risk-based testing approaches and tailoring their usages depending on the context and purpose. The taxonomy classifies different risk drivers, risk assessments, risk-based test processes. All of

these taxonomies provide a generic classification of the applicability of testing techniques in different software engineering projects. However, no taxonomy discusses the dimension of dependencies between code commits and test case types. Classifying these dependencies can potentially aid in the identification and execution of tests that are relevant to the committed code and hence counteract exhaustive testing efforts. The taxonomy presented in this study aims at filling this gap by identifying facets of dependency connections from the viewpoints of software testers.

5.3 Research Method

In this study, we follow the method proposed by Usman et al. [54] to guide the construction of the taxonomy. The method comprises of four phases: i) planning, ii) identification and extraction, iii) design and construction, and iv) validation.

5.3.1 Planning

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

5.3.2 Identification and Extraction

The identification and extraction phase involves identifying the main categories and terms used in the taxonomy. We begin the implementation of this phase by reviewing the literature in search for knowledge about the subject matters. For this purpose, we account for two inclusion criteria in our literature search. First, we wanted to include papers that discuss the impact of specific changes in the code on the quality of the system. Second, we were only interested in papers that were written in English and accessible. The challenge in this phase was to extract terms that are consistent and not interchangeably used in different research studies. Therefore, to overcome this challenge we based our literature search on the set of recognized test case types defined in the international standard ISO/IEC/IEEE CD 2911901:2020(E) document [29] (presented in Section 5.4.1). That is, for each test case type in the ISO document, we searched for relevant papers that empirically investigate or theoretically discuss types of code changes that trigger a reaction among the test cases. The outcome of

Table 5.1: Planning Activities

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

this phase was a list of six categories of code changes and 18 test case types. Further, and based on our literature search, we identified synergy links between the six code categories and the 18 test types (as depicted in Fig 5.2).

5.3.3 Design and Construction

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

5.3.3.1 Survey

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

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

and a category of code change.

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

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

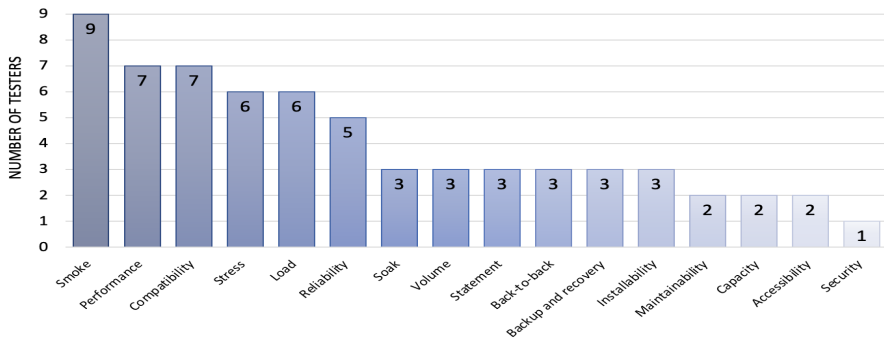


Figure 5.1: Number of Experienced Testers Per Each Test Type.

5.3.3.2 Workshop with Testers

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

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

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

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

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

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

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

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

5.3.4 Validation

This phase ensures that the selected subject matters are clear and thoroughly classified [54]. This can be achieved using three distinct methods: Orthogonality demonstration, benchmarking and utility demonstration. Most of the taxonomies proposed in Software Engineering are evaluated via an utility demonstration, i.e., authors apply their taxonomy to an example [54]. In turn, benchmarking is used to compare the classification capabilities of different taxonomies. In both cases, the taxonomy needs to be applied in actual software artefacts. For this study, we cannot perform those types of validation because we do not have access to test cases or code changes from our industry partners. Therefore, we validate our taxonomy using an orthogonality demonstration. That is, we demonstrate and discuss the orthogonality between strongly dependent categories from the viewpoints of testers. The goal is to illustrate the unique classifications offered by our taxonomy. Based on this demonstration, we aim to highlight which types of tests map to unique types of code changes, as well as those dependencies that cover multiple types of tests.

5.4 Results

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

5.4.1 Test Case Types

In this paper, we decided to base our literature search for extracting code change categories on the list of test case types defined in this ISO/IEC/IEEE CD 2911901:2020(E) document [29]. This was done to overcome the challenge of encountering different terms of test case types that are used interchangeably in published articles. For example, the terms ‘back to back’and ‘differential’testing can be found and used interchangeably in the literature. Table 5.2 lists the definitions of all test case types that we used in our literature search. We used each test case type in the Table to search for relevant papers that empirically investigate or theoretically discuss the dependency between the relevant test case type and code changes.

Table 5.2: Definitions of Test Case Types

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

5.4.2 Code Change Categories and Dependencies with Test Case Types

Our literature search returned a set of 16 relevant papers from which we could extract six different categories of code changes. These categories were: 1) Memory Management, 2) Complexity, 3) Design, 4) Dependency, 5) Conditional, 6) Data. Based on the literature search, we identified 21 dependency links between the six drawn categories of code and eight out of the 18 test case types defined in the ISO document, as shown in Fig 5.2. Each dependency corresponds to a relation where a change in one of the code category results in a failure of a test case of specific type.

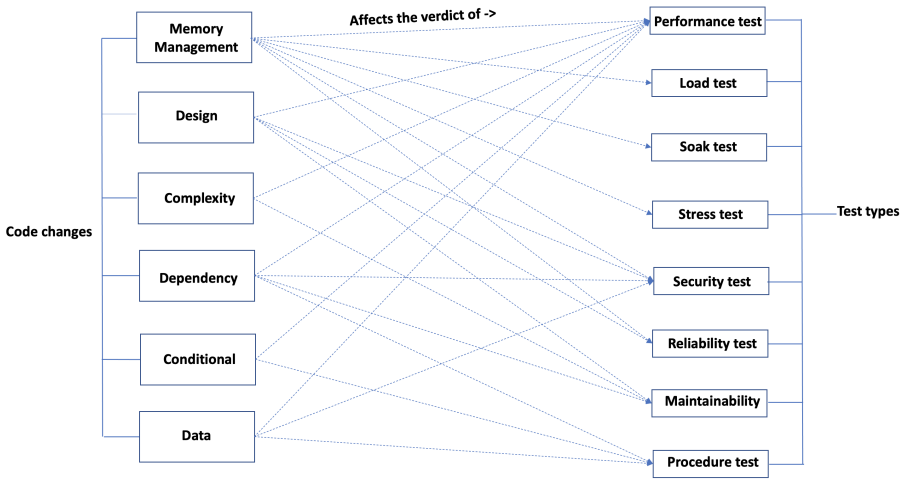


Figure 5.2: Extracted Categories of Code Changes and Their Dependency with Test Case Types.

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

Memory management: This category of change involves groups that are concerned with the management of memory occupied by the system during run-time. Such changes include introducing/fixing memory leaks, buffer overflow, dangling pointers, and resource interference with multi-threading. The following test types would react to this category of change: performance [118], load [119], security [120] [121], soak [122], stress [123], reliability [124] tests. A common memory leak scenario occurs when a developer allocates memory space using the *new* or *malloc* keywords, and misses freeing memory space after they were used. As the program grows in size, less memory becomes available and thereby a performance degradation is encountered. The code example in Fig 5.3 shows how the memory space allocated for pointer `pListElementNext` was unfreed from the memory after being used in revision 2.

Complexity: This category represents changes that add/reduce the time complexity of the program. It includes changes such as adding or removing loops, conditional statements, nesting blocks and/or recursions. The following test types have been identified to react to this category of change: perfor-

Revision 1

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

deleted

Revision 2

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

Figure 5.3: Code Example For Memory Management Change.

mance [125], [126], maintainability [127], [128] tests. Fig 5.4 shows a code example for finding the maximum integer element in an array. The function in the first revision takes a one dimensional array as input, whereas the second revision is modified to accept two-dimensional arrays. The nested loop added to the function in revision 2 would result in an increased time complexity order. Similar changes can potentially trigger performance degradation and thereby performance test failures.

Revision 1

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

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

Revision 2

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

Figure 5.4: Code Example For Complexity Change.

Design: This category involves changes that include code refactoring, adding or removing methods, classes, interfaces, and enumerators, and code smells. The following test types have been identified to react to this category of change: maintainability [127], performance [127], security [129], and reliability [130]. The code example in in Fig 5.5 illustrates a design change in a program that computes the sum of an array elements. The function 'CalculateRank' was added in the modified revision to handle the task of summing up the array elements. Such design decisions reduce the amount of code lines in the program and thus improves its maintainability.

Dependency: This category describes a code change that involves adding/ removing/ modifying a dependency to another module/ library. It can be importing/ removing/ modifying a new library, a new namespace, or a new class. Changes in the dependencies between software artefacts can trigger the following tests: maintainability [131], security [120], procedure [132], and performance [126].

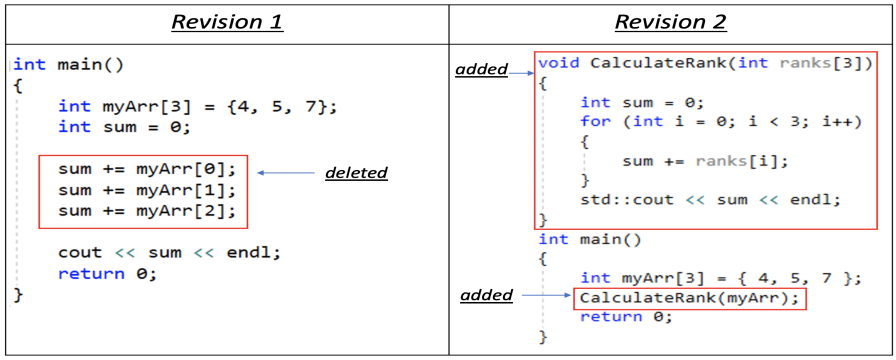


Figure 5.5: A Code Example For Design Change.

Conditional: This category of change occurs when a logical operator or a comparative value in a condition is modified. A misuse in the logical expressions might result in generating the wrong outputs. Performance and procedure tests [126] [132] were identified as dependent to this category of change.
Data change: This category involves 1) changing functions' parameters, 2) passing parameters of incompatible types to modules/ functions, and 3) adding/ fixing assignments of incompatible types to variables, casting statements, and array size allocations, and 4) modifying variable declarations. The following tests would react to such code changes: security [133], performance [126], and procedure [132].

5.4.3 Dependency Patterns and Strengths

5.4.3.1 Survey.

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

- The strongest dependency patterns were mostly concentrated around the memory management and complexity categories of code changes.
- There was a general consensus between the testers about the mappings between performance, soak, load, stress, capacity, and volume tests and the six types of code change categories.
- Most of the discrepancies in the responses were in the classification of the design, dependency, and data categories.

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

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

- Two additional test types, i.e., not found in our literature extraction, were proposed by the testers: Regression and functional tests. The ISO/IEC/IEEE CD 2911901:2020(E) considers these two types of tests as testing activities, since these can be applied at any point in time irrespective of the testing level (unit, integration, system, and user acceptance) [29].

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

5.4.3.2 Workshop

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

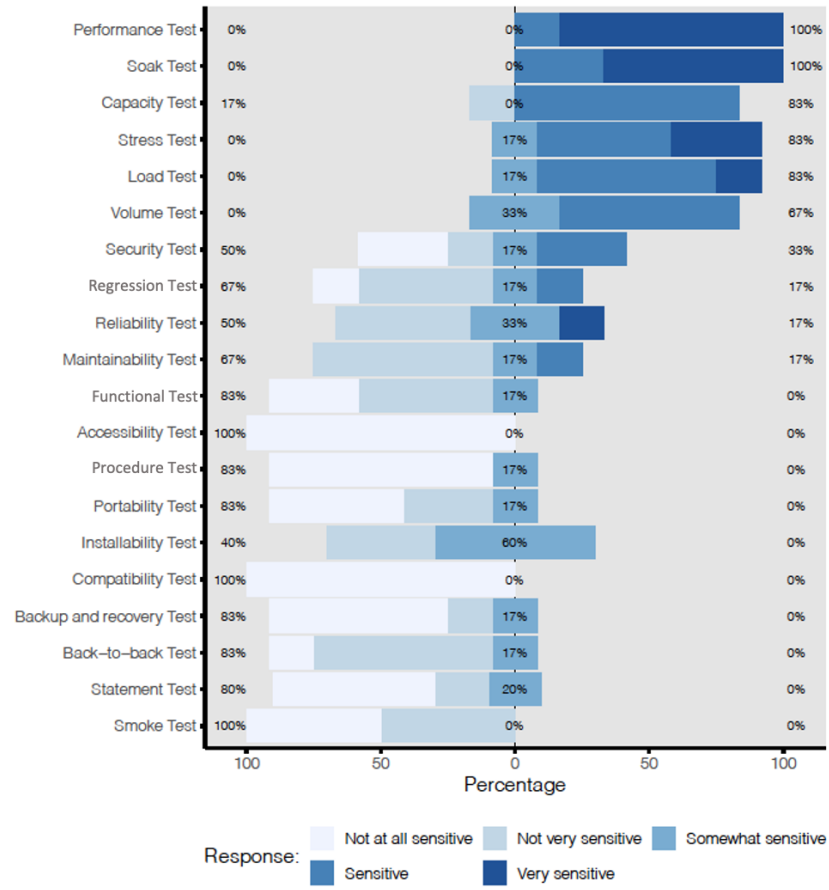


Figure 5.7: Diverging plot showing the strength of perceived connections between each test type and memory management changes. The percentages to the right indicate the proportion of testers that see a stronger relationship, in contrast to those that see a weaker relationship. Testers with a neutral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

5.4.3.3 Memory Management

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

> *"It's not that often that the smoke tests will break due to memory management changes but one possible scenario for this to happen is when we switch from C to C++ first we changed the compiler, then we started modernizing the code*

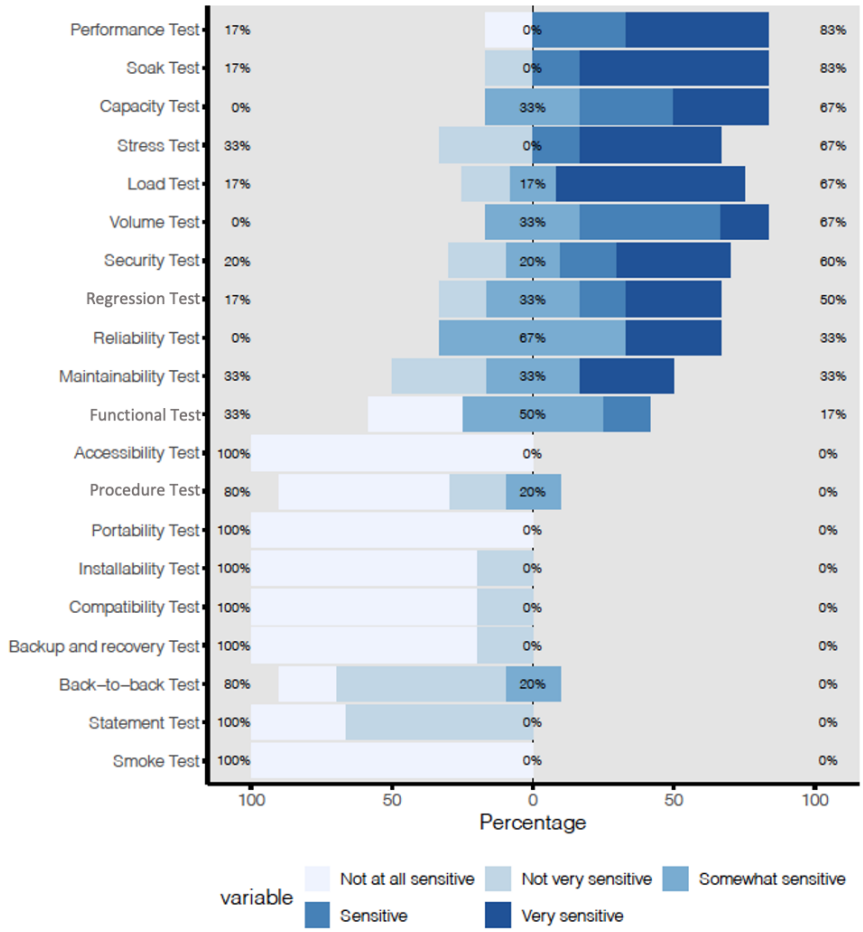


Figure 5.8: Diverging plot showing the strength of perceived connections between each test type and complexity changes. The percentages to the right indicate the proportion of testers that see a stronger relationship, in contrast to those that see a weaker relationship. Testers with a neutral view are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

> *to use smart pointers. Another scenario is when we do major refactoring to optimize the code base."* — Participant 1

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

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

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

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

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

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

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

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

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

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

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

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

5.4.3.4 Complexity code changes

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

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

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

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

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

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

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

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

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

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

5.4.4 Resulting Taxonomy

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

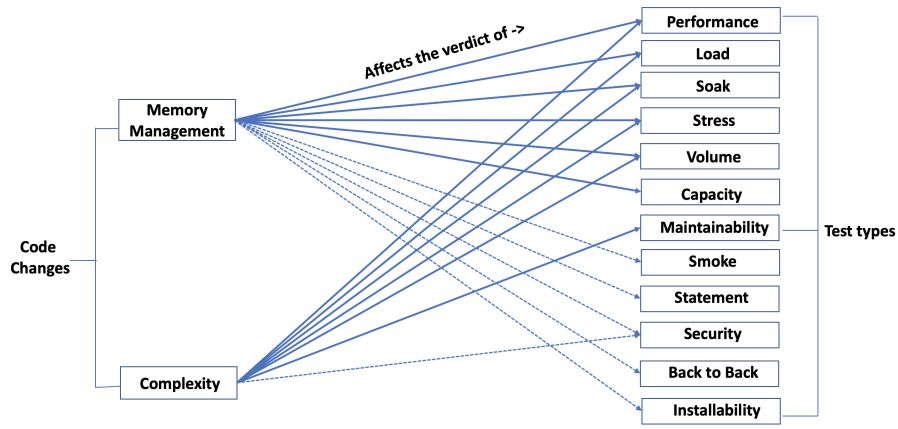


Figure 5.9: The final taxonomy of code changes and test case types. The solid connectors represent strong dependencies perceived by practitioners, whereas the dashed connectors correspond to those dependencies perceived as weak.

5.5 Taxonomy Validation

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

of our taxonomy in relation to automated prediction of relationships between code and tests to support effective test orchestration.

5.5.1 Orthogonality of the Taxonomy's Facets

The majority of relationships are connected to the memory management code changes (11/18). That is not surprising as most of the types of tests found in literature cover system qualities. In fact, during workshops, practitioners rarely mention updates in functionalities (e.g., system requirements), except when discussing complexity changes. Memory management is exclusively connected with 5 test types, such that only 1 of those connections is strong (capacity tests). Consequently, those weak connections can be used to avoid overhead in test executions when focusing the verification of changes in memory management of software systems. Changes in complexity have fewer connections and most of them are actually shared with memory management (6/7), hence indicating a confounding factor between verifying changes in complexity to their impact on verifying memory management. Maintainability is only associated with complexity which is not surprising, since the complexity of a source code has impact on core aspects of maintainability such as testability and debugging [134]. The results shows one weak connection shared between both types of code changes, which is related to security testing. Still, practitioners did not seem to have a consensus on how to handle security tests. Note that on Figs 5.3 and 5.4, security is ranked in the middle between the more explicit agreement and disagreements for both code categories. These contrasting views from practitioners on the purpose of security tests align with the findings drawn by Morrison et al. [135], where the authors highlighted a number of factors that impede the construction of effective vulnerability ML models.

5.5.2 Instrumenting Prediction of Dependencies

Table 5.3 breaks down memory management and complexity changes into specific types and their connection to specific code constructs. We choose C++ constructs because our study encompasses the domain of embedded systems. Future work aims at expanding the constructs to other programming languages such as Java or Python. Associating these code changes to specific code constructs enables automatic extraction and identification of code changes by using information from control version systems, such as git. The process of identifying and classifying code lines into their relevant categories can be instrumented using, for example, a tokenizer and a lexicon of vocabulary that contains a mapping between code tokens (constructs) and their relevant categories of code. For example, a code line that appears with a combination of the tokens *'delete, free, new, and malloc'* can be used to classify a code line as memory management related, since these tokens are used during objects' creation/destruction (Table 5.3). In contrast, automatically identifying and extracting types of tests is more challenging because those tests are used across different levels (e.g., unit or system) such that keyword extraction is inaccurate, particularly for higher levels of testing where tests are written in natural language (e.g., acceptance tests). Therefore, for this study, we assume that practitioners have access to the types of their tests, as part of their test process.

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

Memory Management		
Subcategory	**Description**	**Code Constructs**
Dangling/ Wild pointers	occurs when deleting an object from memory without altering the pointer that points to the object's location.	&variable, *variable, NULL, free
Memory leaks	occur when memory space is allocated but not freed. If such incidents occur, leaks will happen and could eventually cause the program to run out of memory resulting in a program halt.	delete, free, new, malloc
Buffer overflow	occurs when the data gets written past the boundaries of the buffer allocated in memory.	malloc, strcpy, gets, strcmp
Complexity		
Subcategory	**Description**	**Code Constructs**
Loops and conditions	repeating a sequence of instructions for n times until one or more conditions are satisfied. The repetition can occur in the form of multiple nested loops.	for, while, do, if, switch, case, break
recursion	Occurs when a function calls itself until an exit condition is satisfied.	

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

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

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

5.6 Threats to validity

In this section, we briefly discuss the limitations of our paper using the framework recommended by Wohlin et al. [55].

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

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

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

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

5.7 Conclusion and Future Work

The taxonomy presented in this paper aims at classifying dependencies between categories of code changes and test case types. Exploring these dependencies can potentially contribute to the improvement of ML based test case selection approaches that use code analysis and test execution results. In this paper, we have observed strong dependencies between two categories of code changes and seven test case types. This knowledge can gear the test orchestration efforts by pinpointing and executing test cases that are in relation with the relevant changes in the source code. The strongest dependencies were captured between performance, load, stress, soak, volume and the two categories of code changes: memory management and complexity. On the opposite end of the spectrum, the weakest dependencies were found between smoke, back-to-back, installability, accessibility, portability, compatibility, and backup and recovery tests, and the two categories of code changes. Those test cases can be excluded from the suite when the tested code contains memory management and complexity changes

only. As a future work, we plan to continue working on refining the presented taxonomy by investigating additional dependency patterns between other test case types and categories of code changes. Another important future work is to investigate potential dependency links between test script constructs and test execution outcomes of different types. Finally, we aim at evaluating the taxonomy presented in this study by using utility demonstrations on different software projects and programming languages.

Bibliography

[1] A. Brand, L. Allen, M. Altman, M. Hlava, and J. Scott, "Beyond authorship: attribution, contribution, collaboration, and credit," *Learned Publishing*, vol. 28, no. 2, pp. 151–155, 2015.

[2] P. M. Duvall, S. Matyas, and A. Glover, *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[3] E. Knauss, M. Staron, W. Meding, O. Söder, A. Nilsson, and M. Castell, "Supporting continuous integration by code-churn based test selection," in *Proceedings of the Second International Workshop on Rapid Continuous Software Engineering*. IEEE Press, 2015, pp. 19–25.

[4] 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.

[5] G. Rothermel and M. J. Harrold, "A safe, efficient algorithm for regression test selection," in *1993 Conference on Software Maintenance*. IEEE, 1993, pp. 358–367.

[6] J. A. Lee and X. He, "A methodology for test selection," *Journal of Systems and Software*, vol. 13, no. 3, pp. 177–185, 1990.

[7] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. A. London, "Incremental regression testing," in *1993 Conference on Software Maintenance*. IEEE, 1993, pp. 348–357.

[8] R. Gupta, M. J. Harrold, and M. L. Soffa, "An approach to regression testing using slicing," in *Proceedings Conference on Software Maintenance 1992*. IEEE, 1992, pp. 299–308.

[9] V. Antinyan, J. Derehag, A. Sandberg, and M. Staron, "Mythical unit test coverage," *IEEE Software*, vol. 35, no. 3, pp. 73–79, 2018.

[10] F. Wedyan, D. Alrmuny, and J. M. Bieman, "The effectiveness of automated static analysis tools for fault detection and refactoring prediction," in *2009 International Conference on Software Testing Verification and Validation*. IEEE, 2009, pp. 141–150.

[11] C. Bolduc, "Lessons learned: Using a static analysis tool within a continuous integration system," in *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2016, pp. 37–40.

[12] C. Couto, J. E. Montandon, C. Silva, and M. T. Valente, "Static correspondence and correlation between field defects and warnings reported by a bug finding tool," *Software Quality Journal*, vol. 21, no. 2, pp. 241–257, 2013.

[13] S. Kim and M. D. Ernst, "Which warnings should i fix first?" in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2007, pp. 45–54.

[14] S. Yoo, R. Nilsson, and M. Harman, "Faster fault finding at google using multi objective regression test optimisation," in *8th European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'11), Szeged, Hungary*, 2011.

[15] G. A. Liebchen, "Data cleaning techniques for software engineering data sets," Ph.D. dissertation, Brunel University, School of Information Systems, Computing and Mathematics, 2010.

[16] D. F. Nettleton, A. Orriols-Puig, and A. Fornells, "A study of the effect of different types of noise on the precision of supervised learning techniques," *Artificial intelligence review*, vol. 33, no. 4, pp. 275–306, 2010.

[17] 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.

[18] 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.

[19] 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, pp. 6–pp.

[20] 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.

[21] E. N. Narciso, M. E. Delamaro, and F. D. L. D. S. Nunes, "Test case selection: A systematic literature review," *International Journal of Software Engineering and Knowledge Engineering*, vol. 24, no. 04, pp. 653–676, 2014.

[22] R. D. De Veaux and D. J. Hand, "How to lie with bad data," *Statistical Science*, vol. 20, no. 3, pp. 231–238, 2005.

[23] J. Zhang and Y. Yang, "Robustness of regularized linear classification methods in text categorization," in *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, 2003, pp. 190–197.

[24] J. Abellán and A. R. Masegosa, "Bagging decision trees on data sets with classification noise," in *International Symposium on Foundations of Information and Knowledge Systems*. Springer, 2010, pp. 248–265.

[25] 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.

[26] 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.

[27] 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.

[28] C.-M. Teng, "Correcting noisy data." in *ICML*. Citeseer, 1999, pp. 239–248.

[29] "Iso/iec/ieee international standard - software and systems engineering –software testing–part 1: Concepts and definitions," Tech. Rep., 2020.

[30] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov, "An extensive study of static regression test selection in modern software evolution," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 583–594.

[31] A. Shi, M. Hadzi-Tanovic, L. Zhang, D. Marinov, and O. Legunsen, "Reflection-aware static regression test selection," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–29, 2019.

[32] L. Zhang, "Hybrid regression test selection," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 199–209.

[33] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE transactions on software engineering*, vol. 28, no. 2, pp. 159–182, 2002.

[34] A. Srivastava and J. Thiagarajan, "Effectively prioritizing tests in development environment," in *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, 2002, pp. 97–106.

[35] F. G. de Oliveira Neto, A. Ahmad, O. Leifler, K. Sandahl, and E. Enoiu, "Improving continuous integration with similarity-based test case selection," in *Proceedings of the 13th International Workshop on Automation of Software Test*, 2018, pp. 39–45.

[36] H. Hemmati, A. Arcuri, and L. Briand, "Achieving scalable model-based testing through test case diversity," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 1, pp. 1–42, 2013.

[37] 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, Jan 2004. [Online]. Available: https://doi.org/10.1023/A:1025832930864

[38] 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.

[39] 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.

[40] T. Y. Chen and M. F. Lau, "Dividing strategies for the optimization of a test suite," *Information Processing Letters*, vol. 60, no. 3, pp. 135–141, 1996.

[41] G. Rothermel and M. J. Harrold, "A framework for evaluating regression test selection techniques," in *Proceedings of 16th International Conference on Software Engineering.* IEEE, 1994, pp. 201–210.

[42] M. J. Harrold, "Testing evolving software," *Journal of Systems and Software*, vol. 47, no. 2-3, pp. 173–181, 1999.

[43] 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.

[44] M. Pechenizkiy, A. Tsymbal, S. Puuronen, and O. Pechenizkiy, "Class noise and supervised learning in medical domains: The effect of feature extraction," in *19th IEEE symposium on computer-based medical systems (CBMS'06).* IEEE, 2006, pp. 708–713.

[45] N. Juristo and A. M. Moreno, *Basics of software engineering experimentation.* Springer Science & Business Media, 2013.

[46] J. Yao and M. Shepperd, "Assessing software defection prediction performance: Why using the matthews correlation coefficient matters," in *Proceedings of the Evaluation and Assessment in Software Engineering*, 2020, pp. 120–129.

[47] "Dealing with outliers in machine learning," Sep 2021. [Online]. Available: https://expressanalytics.com/blog/outliers-machine-learning/

[48] H. J. Escalante, "A comparison of outlier detection algorithms for machine learning," in *Proceedings of the International Conference on Communications in Computing*, 2005, pp. 228–237.

[49] R. Domingues, M. Filippone, P. Michiardi, and J. Zouaoui, "A comparative evaluation of outlier detection algorithms: Experiments and analyses," *Pattern Recognition*, vol. 74, pp. 406–421, 2018.

[50] M. Petrovskiy, "Outlier detection algorithms in data mining systems," *Programming and Computer Software*, vol. 29, no. 4, pp. 228–237, 2003.

[51] L. H. Chiang, R. J. Pell, and M. B. Seasholtz, "Exploring process data with the use of robust outlier detection algorithms," *Journal of Process Control*, vol. 13, no. 5, pp. 437–449, 2003.

[52] D. Guan, W. Yuan, and L. Shen, "Class noise detection by multiple voting," in *2013 Ninth International Conference on Natural Computation (ICNC)*. IEEE, 2013, pp. 906–911.

[53] B. Sluban and N. Lavrač, "Relating ensemble diversity and performance: A study in class noise detection," *Neurocomputing*, vol. 160, pp. 120–131, 2015.

[54] 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.

[55] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[56] D. Ståhl and J. Bosch, "Experienced benefits of continuous integration in industry software product development: A case study," in *The 12th IASTED International Conference on Software Engineering,(Innsbruck, Austria, 2013)*, 2013, pp. 736–743.

[57] G. Çalikli, M. Staron, and W. Meding, "Measure early and decide fast: transforming quality management and measurement to continuous deployment," in *Proceedings of the 2018 International Conference on Software and System Process*. ACM, 2018, pp. 51–60.

[58] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of the 27th international conference on Software engineering*. ACM, 2005, pp. 284–292.

[59] F. Chollet, *Deep Learning with Python*. Manning, 2017.

[60] A. Géron, *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. Oreilly, 2015.

[61] R. Saxena, "Introduction to decision tree algorithm," 2017. [Online]. Available: https://dataaspirant.com/2017/01/30/how-decision-tree-algorithm-works/

[62] M. Awad and R. Khanna, *Efficient learning machines: theories, concepts, and applications for engineers and system designers*. Apress, 2017.

[63] I. Gondra, "Applying machine learning to software fault-proneness prediction," *Journal of Systems and Software*, vol. 81, no. 2, pp. 186–195, 2008.

[64] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Transactions on Software Engineering*, vol. 37, no. 6, pp. 772–787, Nov 2011.

[65] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Transactions on Software Engineering*, vol. 26, no. 7, pp. 653–661, July 2000.

[66] J. Beningo, "Using the static keyword in c," https://community.arm.com/developer/ip-products/system/b/embedded-blog/posts/using-the-static-keyword-in-c, 2014.

[67] V. H. Durelli, R. S. Durelli, S. S. Borges, A. T. Endo, M. M. Eler, D. R. Dias, and M. P. Guimarães, "Machine learning applied to software testing: A systematic mapping study," *IEEE Transactions on Reliability*, 2019.

[68] B. Busjaeger and T. Xie, "Learning for test prioritization: an industrial case study," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 975–980.

[69] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, and B. Xie, "Learning to prioritize test programs for compiler testing," in *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 2017, pp. 700–711.

[70] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement learning for automatic test case prioritization and selection in continuous integration," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2017, pp. 12–22.

[71] M. Azizi and H. Do, "A collaborative filtering recommender system for test case prioritization in web applications," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, ser. SAC '18. New York, NY, USA: ACM, 2018, pp. 1560–1567. [Online]. Available: http://doi.acm.org/10.1145/3167132.3167299

[72] F. Palma, T. Abdou, A. Bener, J. Maidens, and S. Liu, "An improvement to test case failure prediction in the context of test case prioritization," in *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*, ser. PROMISE'18. New York, NY, USA: ACM, 2018, pp. 80–89. [Online]. Available: http://doi.acm.org/10.1145/3273934.3273944

[73] T. B. Noor and H. Hemmati, "Studying test case failure prediction for test case prioritization," in *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*. ACM, 2017, pp. 2–11.

[74] ——, "A similarity-based approach for test case prioritization using historical failure data," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2015, pp. 58–68.

[75] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[76] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[77] K. W. Al-Sabbagh, M. Staron, R. Hebig, and W. Meding, "Predicting test case verdicts using textual analysis of committed code churns," in *Joint Proceedings of the International Workshop on Software Measurementand the International Conference on Software Process and Product Measurement (IWSM Mensura 2019)*, vol. 2476, 2019, pp. 138–153.

[78] H. Hata, O. Mizuno, and T. Kikuno, "Fault-prone module detection using large-scale text features based on spam filtering," *Empirical Software Engineering*, vol. 15, no. 2, pp. 147–165, 2010.

[79] S. Kim, E. J. Whitehead Jr, and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 181–196, 2008.

[80] L. Aversano, L. Cerulo, and C. Del Grosso, "Learning from bug-introducing changes to prevent fault prone code," in *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*. ACM, 2007, pp. 19–26.

[81] G. H. John, "Robust decision trees: Removing outliers from databases." in *KDD*, vol. 95, 1995, pp. 174–179.

[82] Q. Zhao and T. Nishida, "Using qualitative hypotheses to identify inaccurate data," *Journal of Artificial Intelligence Research*, vol. 3, pp. 119–145, 1995.

[83] J. A. Sáez, J. Luengo, and F. Herrera, "Evaluating the classifier behavior with noisy data considering performance and robustness: The equalized loss of accuracy measure," *Neurocomputing*, vol. 176, pp. 26–35, 2016.

[84] X. Zhu and X. Wu, "Class noise vs. attribute noise: A quantitative study," *Artificial intelligence review*, vol. 22, no. 3, pp. 177–210, 2004.

[85] O. Mizuno, S. Ikami, S. Nakaichi, and T. Kikuno, "Spam filter based approach for finding fault-prone software modules," in *Proceedings of the Fourth International Workshop on Mining Software Repositories*. IEEE Computer Society, 2007, p. 4.

[86] S. Boughorbel, F. Jarray, and M. El-Anbari, "Optimal classifier for imbalanced data using matthews correlation coefficient metric," *PloS one*, vol. 12, no. 6, 2017.

[87] B. Frénay and M. Verleysen, "Classification in the presence of label noise: a survey," *IEEE transactions on neural networks and learning systems*, vol. 25, no. 5, pp. 845–869, 2013.

[88] E. Knauss, S. Houmb, K. Schneider, S. Islam, and J. Jürjens, "Supporting requirements engineers in recognising security issues," in *International Working Conference on Requirements Engineering: Foundation for Software Quality*. Springer, 2011, pp. 4–18.

[89] M. Ochodek, R. Hebig, W. Meding, G. Frost, and M. Staron, "Recognizing lines of code violating company-specific coding guidelines using machine learning," *Empirical Software Engineering*, vol. 25, no. 1, pp. 220–265, 2020.

[90] H. Sajnani, "Automatic software architecture recovery: A machine learning approach," in *2012 20th IEEE International Conference on Program Comprehension (ICPC)*. IEEE, 2012, pp. 265–268.

[91] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 297–308.

[92] Z. Cai, L. Lu, and S. Qiu, "An abstract syntax tree encoding method for cross-project defect prediction," *IEEE Access*, vol. 7, pp. 170 844–170 853, 2019.

[93] K. W. Al-Sabbagh, M. Staron, R. Hebig, and W. Meding, "Improving data quality for regression test selection by reducing annotation noise," in *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2020, pp. 191–194.

[94] T. Zimmermann and P. Weißgerber, "Preprocessing cvs data for fine-grained analysis." in *MSR*, vol. 4, 2004, pp. 2–6.

[95] C. M. Teng, "Combining noise correction with feature selection," in *International Conference on Data Warehousing and Knowledge Discovery*. Springer, 2003, pp. 340–349.

[96] K. W. Al-Sabbagh, R. Hebig, and M. Staron, "The effect of class noise on continuous test case selection: A controlled experiment on industrial data," in *International Conference on Product-Focused Software Process Improvement*. Springer, 2020, pp. 287–303.

[97] T. M. Khoshgoftaar and J. Van Hulse, "Empirical case studies in attribute noise detection," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 39, no. 4, pp. 379–388, 2009.

[98] C.-M. Teng, "A comparison of noise handling techniques." in *FLAIRS Conference*, 2001, pp. 269–273.

[99] J. R. Quinlan, "Induction of decision trees," *Machine learning*, vol. 1, no. 1, pp. 81–106, 1986.

[100] C. E. Brodley and M. A. Friedl, "Identifying mislabeled training data," *Journal of artificial intelligence research*, vol. 11, pp. 131–167, 1999.

[101] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 181–190.

[102] S. Amasaki, Y. Takagi, O. Mizuno, and T. Kikuno, "A bayesian belief network for assessing the likelihood of fault content," in *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.* IEEE, 2003, pp. 215–226.

[103] J. Deng, L. Lu, S. Qiu, and Y. Ou, "A suitable ast node granularity and multi-kernel transfer convolutional neural network for cross-project defect prediction," *IEEE Access*, vol. 8, pp. 66 647–66 661, 2020.

[104] T. B. C. Arias, P. Avgeriou, and P. America, "Analyzing the actual execution of a large software-intensive system for determining dependencies," in *2008 15th Working Conference on Reverse Engineering.* IEEE, 2008, pp. 49–58.

[105] A. Hamou-Lhadj and T. C. Lethbridge, "A survey of trace exploration tools and techniques," in *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, 2004, pp. 42–55.

[106] M. Balint, R. Marinescu, and T. Girba, "How developers copy," in *14th IEEE International Conference on Program Comprehension (ICPC'06).* IEEE, 2006, pp. 56–68.

[107] V. Ganganwar, "An overview of classification algorithms for imbalanced datasets," *International Journal of Emerging Technology and Advanced Engineering*, vol. 2, no. 4, pp. 42–47, 2012.

[108] A. Axelrod, *Complete Guide to Test Automation.* Springer, 2018.

[109] K. Wang, C. Zhu, A. Celik, J. Kim, D. Batory, and M. Gligoric, "Towards refactoring-aware regression test selection," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE).* IEEE, 2018, pp. 233–244.

[110] B. H. Kwasnik, "The role of classification in knowledge representation and discovery," 1999.

[111] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong, "Orthogonal defect classification-a concept for in-process measurements," *IEEE Transactions on software Engineering*, vol. 18, no. 11, pp. 943–956, 1992.

[112] N. Li, Z. Li, and X. Sun, "Classification of software defect detected by black-box testing: An empirical study," in *2010 Second World Congress on Software Engineering*, vol. 2. IEEE, 2010, pp. 234–240.

[113] L. Ma and J. Tian, "Web error classification and analysis for reliability improvement," *Journal of Systems and Software*, vol. 80, no. 6, pp. 795–804, 2007.

[114] R. Britto, "Knowledge classification for supporting effort estimation in global software engineering projects," Ph.D. dissertation, Blekinge Tekniska Högskola, 2015.

[115] J. Novak, A. Krajnc *et al.*, "Taxonomy of static code analysis tools," in *The 33rd International Convention MIPRO*. IEEE, 2010, pp. 418–422.

[116] S. Vegas, N. Juristo, and V. R. Basili, "Maturing software engineering knowledge through classifications: A case study on unit testing techniques," *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 551–565, 2009.

[117] M. Felderer and I. Schieferdecker, "A taxonomy of risk-based testing," *arXiv preprint arXiv:1912.11519*, 2019.

[118] Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *Proceedings of the 36th international conference on software engineering*, 2014, pp. 1013–1024.

[119] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automatic identification of load testing problems," in *2008 IEEE International Conference on Software Maintenance*. IEEE, 2008, pp. 307–316.

[120] F. Cohen, "Information system attacks: A preliminary classification scheme," *Computers & Security*, vol. 16, no. 1, pp. 29–46, 1997.

[121] R. C. Seacord and A. D. Householder, "A structured approach to classifying security vulnerabilities," CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, Tech. Rep., 2005.

[122] T. Karttunen, "Implementing soak testing for an access network solution," Ph.D. dissertation, HELSINKI UNIVERSITY OF TECHNOLOGY, 2009.

[123] J. Zhang, S.-C. Cheung, and S. T. Chanson, "Stress testing of distributed multimedia software systems," in *Formal Methods for Protocol Engineering and Distributed Systems*. Springer, 1999, pp. 119–133.

[124] D. Cotroneo, R. Pietrantuono, L. Mariani, and F. Pastore, "Investigation of failure causes in workload-driven reliability testing," in *Fourth international workshop on Software quality assurance: in conjunction with the 6th ESEC/FSE joint meeting*, 2007, pp. 78–85.

[125] A. Nistor, P.-C. Chang, C. Radoi, and S. Lu, "Caramel: Detecting and fixing performance problems that have non-intrusive fixes," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 902–912.

[126] J. P. Sandoval Alcocer, A. Bergel, and M. T. Valente, "Learning from source code history to identify performance failures," in *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, 2016, pp. 37–48.

[127] L. Jiang, Z. Su, and E. Chiu, "Context-based detection of clone-related bugs," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2007, pp. 55–64.

[128] R. D. Banker, S. M. Datar, and D. Zweig, "Software complexity and maintainability," *Age*, vol. 11, no. 5.6, p. 3, 1989.

[129] T. Aslam, "A taxonomy of security faults in the unix operating system," *Master's thesis, Purdue University*, vol. 199, no. 5, 1995.

[130] Y. Levendel, "Defects and reliability analysis of large software systems: field experience," in *1989 The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers.* IEEE Computer Society, 1989, pp. 238–239.

[131] E. Razina and D. S. Janzen, "Effects of dependency injection on maintainability," in *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications: Cambridge, MA*, 2007, p. 7.

[132] A. Sawant, P. H. Bari, and P. Chawan, "Software testing techniques and strategies," 2012.

[133] Z. Yan, D. Guowei, G. Tao, and Y. Jianyu, "Taxonomy of source code security defects based on three-dimension-tree," in *International Conference on Computer and Computing Technologies in Agriculture.* Springer, 2013, pp. 232–241.

[134] M. Felderer, B. Marculescu, F. G. de Oliveira Neto, R. Feldt, and R. Torkar, "A testability analysis framework for non-functional properties," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW).* IEEE, 2018, pp. 54–58.

[135] P. Morrison, K. Herzig, B. Murphy, and L. Williams, "Challenges with applying vulnerability prediction models," in *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*, 2015, pp. 1–9.