Improving Test Case Selection by Handling Class and Attribute Noise

Khaled Al-Sabbagh, Miroslaw Staron and Regina Hebig

September 13, 2021

Improving Test Case Selection By Handling Class and Attribute Noise

Khaled Walid Al-Sabbagh*, Miroslaw Staron, Regina Hebig

Chalmers | University of Gothenburg Gothenburg, Sweden

Abstract

Big data and machine learning models have been increasingly used to support software engineering 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 performances. In this paper, we address this issue by studying the effect of two types of noise (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 and removing duplicate ones. 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% f-score compared with 44% precision, 17% recall, and 25% f-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 could be drawn.

Keywords:
Class Noise, Attribute Noise, Test Case Selection
2020 MSC: 68N30

*Corresponding author

Email addresses: khaled.al-sabbagh@gu.se (Khaled Walid Al-Sabbagh), miroslaw.staron@gu.se (Miroslaw Staron), regina.hebig@gu.se (Regina Hebig)

1. Introduction

Machine Learning (ML) models have been increasingly used for automating software engineering activities [1, 2, 3, 4]. 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. Since regression testing is performed frequently (after every commit), they often 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 1 illustrates a CI pipeline that includes a number of accrued test suites of varying sizes. 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 seeks to identify and select a subset of test cases from the pool of available tests to perform regression testing. These test cases are then added to the every-build suite and get executed as soon as new code check-ins are built. Failure to detect faults in this early phase of testing will prolong faults' discovery until larger suites (the daily or weekend suites) are executed.

This way of orchestrating test cases increases the development speed and reduces the cost of regression testing as faults are continuously discovered and fed-back to developers in a quick manner. Figure 1 exemplifies a scenario where the CI system misses including test case 2 (*Tc2*) in the every-build suite for execution. Failure to include (*Tc2*) in the every-build suite is penalized by a prolonged time and cost of testing. Therefore, finding an effective approach for test case selection (TCS) is important to maximize the probability of detecting faults as soon 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 [5] [6], static code metrics [7][8], natural language processing (NLP) [9][10]. These approaches use data-sets with historical defects for training machine learning (ML) models to classify code as either non-defective or defective (i.e. in need for testing) or to predict whether test cases will fail. In our previous work [9], 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

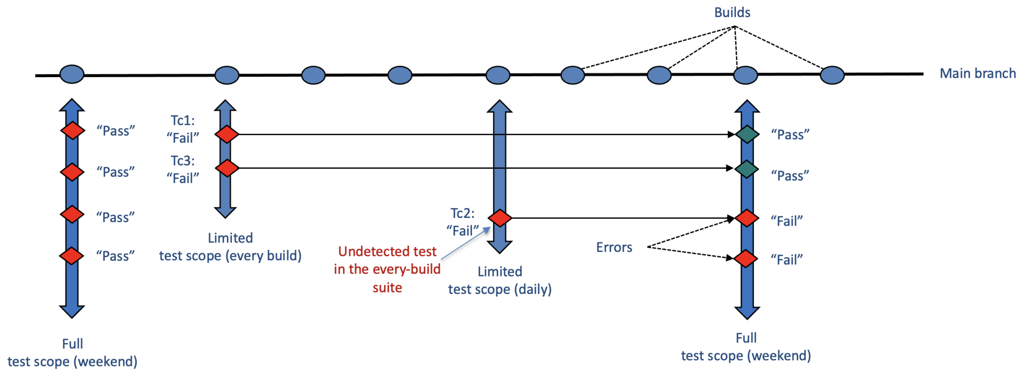


Figure 1: CI Pipeline with Test Case Selection.

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[11]. However, two categories of noise types are most commonly addressed in the literature - class and attribute noise [12, 13, 14, 15]. Class noise (also known as label or annotation noise) occurs as a result of either contradictory entries or mislabeling training entries [15], 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 [15][16].

In the domain 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 predictors and would consequently hamper their classification accuracy. Similarly, one example of attribute noise in the domain of TCS can occur when similar lines in the code are written using two or more coding styles. Written code lines in the less frequently used style will be characterized by attributes whose frequency deviates from similar code lines written in the more recurring style(s). These deviations in the frequency of attributes between similar code lines can negatively affect the learning performance.

A number of research studies proposed several techniques for handling class and attribute noise [12][17][18][19]. 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[10], we introduced an approach for addressing the problem of annotation noise by relabeling contradictory entries and removing duplicate entries that belong to one of the binary classes. The empirical evaluation of applying the technique was measured with respect to precision, recall, and f-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) [20], on the performance of a TCS learner using the class-noise cleaned data reported in [10]. The purpose is to provide testers with insights into choosing the right noise handling strategies, and consequently achieve more effective TCS. For this purpose, we design and implement a controlled experiment on the same industrial data-set used in our previous study [10] 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 f-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.

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 [21]. 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.

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 this paper, we mention some of these previous work, as summarized in Table 1

A previous work on test case selection [9] 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. [22] used text mining and spam filtering algorithms to classify software modules into

Study	Type	Systems	Predictors	Results
[22]	Defects Prediction	BIRT, ECLP, MODE, TPTP, and WTP	Spam Filter	Precision 40, Recall 80
[23]	Defects Prediction	ArgoUML and BIRT	Spam Filter	Precision 72, Recall 70 Precision75, Recall 72
[24]	Defects Prediction	JHotDraw and DNS	Regression, ADABoosting, C4.5, SVM, K-NN	K-NN: Precision 59, Recall 69 Precision 59, Recall 23
[2]	Defects Prediction	Apache 1.3, Bugzilla, Columba, Gaim, GForge, JEdit, Mozilla, Eclipse JDT, Plone, PostgreSQL, Scarab, and Subversion	SVM	Accuracy 78, Recall 60
[9]	TCS	Industrial Software	RF	Precision 73, Recall 48

Table 1: Results Summary For Defects Prediction and Test case Selection Research

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.[23] 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. [24] 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. [2] 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.

2.2. Class Noise Handling Research

Brodley et al.[17] 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.[25] 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, f-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.[12] introduced CFAUD, a variant of the approach proposed by Brodley et al.[17], 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.[13] 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.

2.3. Attribute Noise Handling Research

Khoshgoftaar et al.[11] 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 [17]. 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. [18] 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 [17] was made. The results reported different effectiveness scores ranging from 24% to 100% effectiveness.

Khoshgoftaar et al.[26] extended their work in [20] 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[27] 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[28] 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[15].

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 [28] [29][15].

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.

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, ..., F_n >. Each function in *P* consists of a sequence of statements *S* <S_1,...,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,...,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, ...> 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.

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 [9]. 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 being 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 [30] 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 3.2 lists and describes some of the most popular approaches for defects and test prediction using source code analysis. The Table also includes a list of some of the advantages and disadvantages of each approach, and a brief comparison between these approaches and MeBoTs.

[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.

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: [31], [32]	**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: [5], [6], [33]	**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: [34], [35]	**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.

Table 2: Comparing Popular Defect and Test Prediction Approaches 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.

3.3. Noise Definitions and Examples

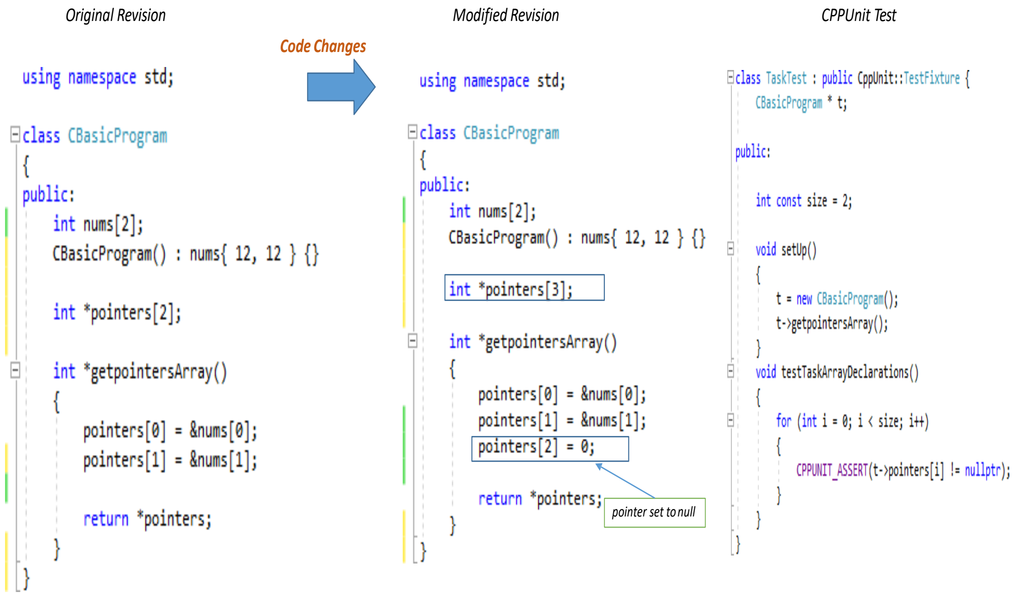
Noisy observations are commonly determined by two factors: 1) the correctness of the class values, and 2) by how well the selected attributes describe learning instances in the training data. This section provides a definition and an example for each type of noise (class and attribute) found when analyzing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

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 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 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.

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



Original Revision

```cpp
using namespace std;

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

    int *pointers[2];

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

        return *pointers;
    }
}
```

Modified Revision

```cpp
using namespace std;

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

    int *pointers[3];

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

        return *pointers;
    }
}
```

CPPUnit Test

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

public:

    int const size = 2;

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

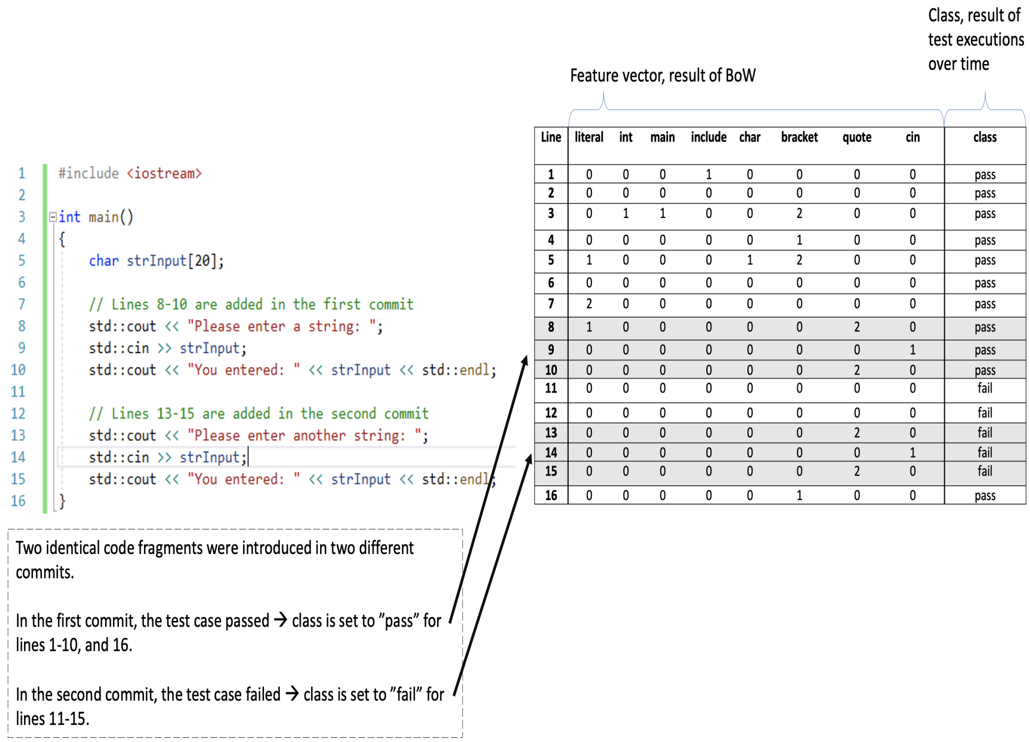
Figure 2: Example On the Relationship Between Code Churns and Test Case Verdicts

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



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

Two identical code fragments were introduced in two different commits.

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

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

Feature vector, result of BoW

Class, result of test executions over time

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 3: Class Noise Example in Code Base.

The feature vectors in Figure 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:

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

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. [20], which suggests that a noisy observation appears when one or more of its attributes deviates from the general distribution of other attributes. The larger the deviation is for one or more observations, the more evidence there is that they are noisy. In the context of the given problem (i.e., TCS), a deviation between attributes can occur when the general distribution of S follows a standard coding style, whereas a smaller fraction of S deviates from the standard.

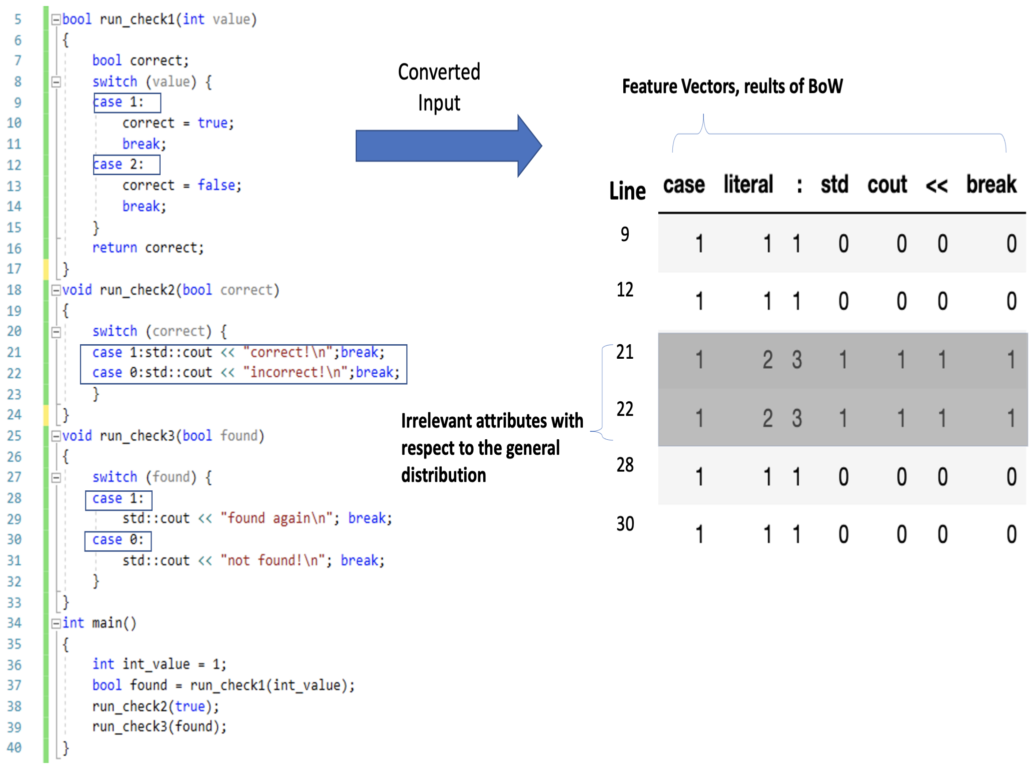
As an illustration of those deviations in code churns, Figure 4 exemplifies two coding styles used for expressing ‘case’ blocks in a C++ program. By examining the ‘case’ blocks in the ‘run_check1’, ‘run_check2’, and ‘run_check3’ functions, we notice that the first and most reoccurring style uses a line 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. 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 [36][15][13]. 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 [10] to handle the problem of class noise. Thereafter, an existing elimination based approach from the literature for handling attribute noise is described.

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 [37], and 2) merge transactions [14]. 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



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

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

Figure 4: Attribute Noise Example in Code Base.

in a quick fashion [37]. 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 [14] 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 class value of the repeated instance in the original set is annotated with 1 (passed) and the class value of the same instance in the dictionary is annotated with 0 (failed), then relabel the class value for the instance in the dictionary from 0 to 1. If the class 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].

4.2. 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 [20] 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

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

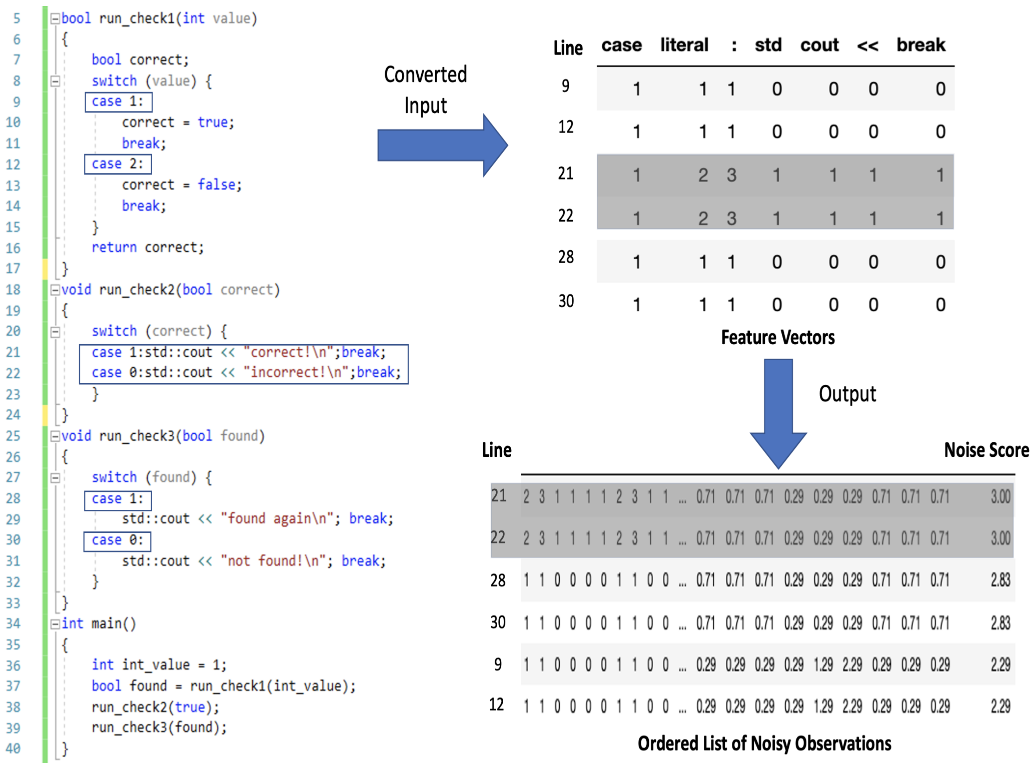
normal. Upon ranking noisy instances, the generated list can be used to toss out instances (code lines) that come with the highest rank with respect to attribute noise.

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

Figure 5 exemplifies the output produced by the PANDA algorithm when applied on the code fragment presented in Section 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 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.

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.



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

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

Feature Vectors

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

Ordered List of Noisy Observations

Figure 5: An Excerpt of PANDA's Output

5.1. Original Data Set

In the baseline paper [10], 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 presented in [10] 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.

5.2. Random Forest For Evaluation

In this study, the MeBoTS method described in Section 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 performance. The performance results produced by the model using an n_estimater of 300 can be found in Appendix A.

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 f-score, for an ML model.

Applying the class noise handling approach (described in Section 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.

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.

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 the class-noise cleaned data.

According to Ganganwar [38], 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) to determine the balance of classes. Figure 6 shows that the distribution of instances in the non-defective class contains many more samples than the defective class (14400 to 5415 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 [39]. 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.

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

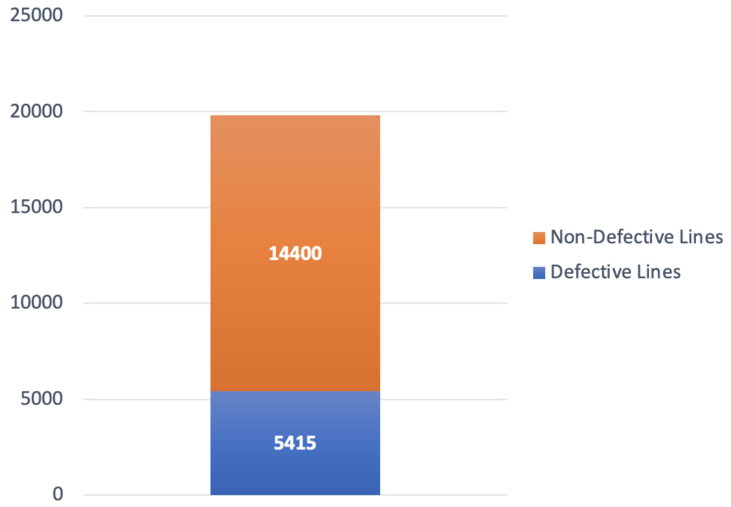


Figure 6: The Distribution of Classes In The Adopted Data-Set

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.

5.4.3. Dependent Variables

The dependent variables are three evaluation measures used for the performance of an ML classifier – Precision, Recall, and F-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 F-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.

5.4.4. Experimental Hypotheses

Three hypotheses are defined according to the goals of this study and tested for statistical significance in Section5.4.5. 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*
$$\mu 1p = \mu 2p \tag{1}$$
.
- *H0r: The mean Recall is the same for a model with and without attribute noise*
$$\mu 1r = \mu 2r \tag{2}$$
.
- *H0f: The mean F-score is the same for a model with and without attribute noise*
$$\mu 1f = \mu 2f \tag{3}$$
.

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.

5.4.5. Data Analysis Methods

The experimental data were analyzed using the scikit learn library [39]. 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 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).

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 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 f-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].

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

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

Table 3: An Excerpt Of the Output Generated From PANDA

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 f-score.

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 7. The Figure shows a non-normalized matrix for the predicted and actual values of test case verdicts for all lines in the test set. The first cell on the upper left hand side corresponds to the number of lines (6,543) that are predicted to trigger test case failures and are actually true. On the same diagonal, the last cell to the bottom right of the matrix indicates the number of lines (15,688) that are predicted to be non-defective and are actually true, and require no testing. The remaining entries in the test set correspond to the number of misclassified lines.

The bar chart in Figure 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% F-score compared to the learning achieved on the original data.

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 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 9, 10, and 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.

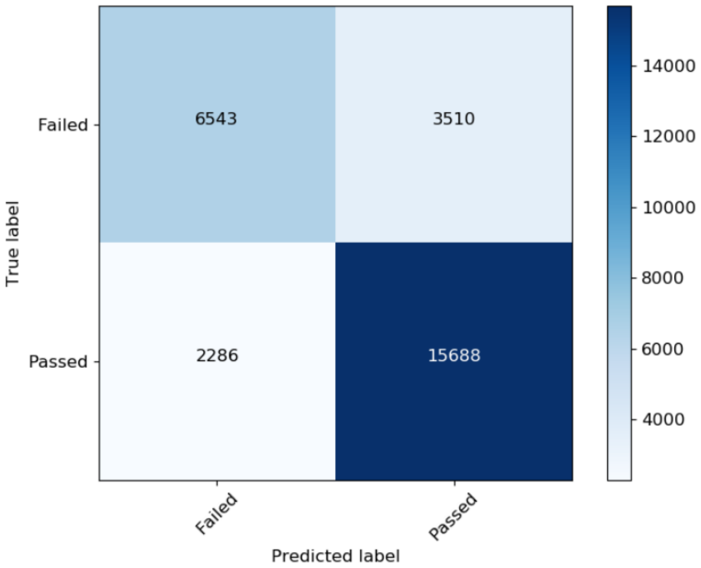


Figure 7: Confusion Matrix For a Classifier Trained on Class Noise cleaned Data

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, 5, and 6. Note that the precision, recall, and f-score reported in the three tables under 0% treatment level are different than those obtained from training on the class-noise cleaned data. This is because the control group was used as a baseline for the class-noise cleaned data from which the ML model was built.

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 12, 13, and 14. Then, the Shapiro-Wilk test was carried out to further support the observations drawn from the graphs. As can be seen from the graphs, the distributions appear to be negatively skewed (asymmetric), and thereby the assumption of normality in the distribution of the three variables do not hold. The Shapiro-Wilk test results supported the observation drawn from the graphs and revealed that the null hypotheses of normality for the three dependent variables can be rejected (p-value <0.05), as shown in Tables

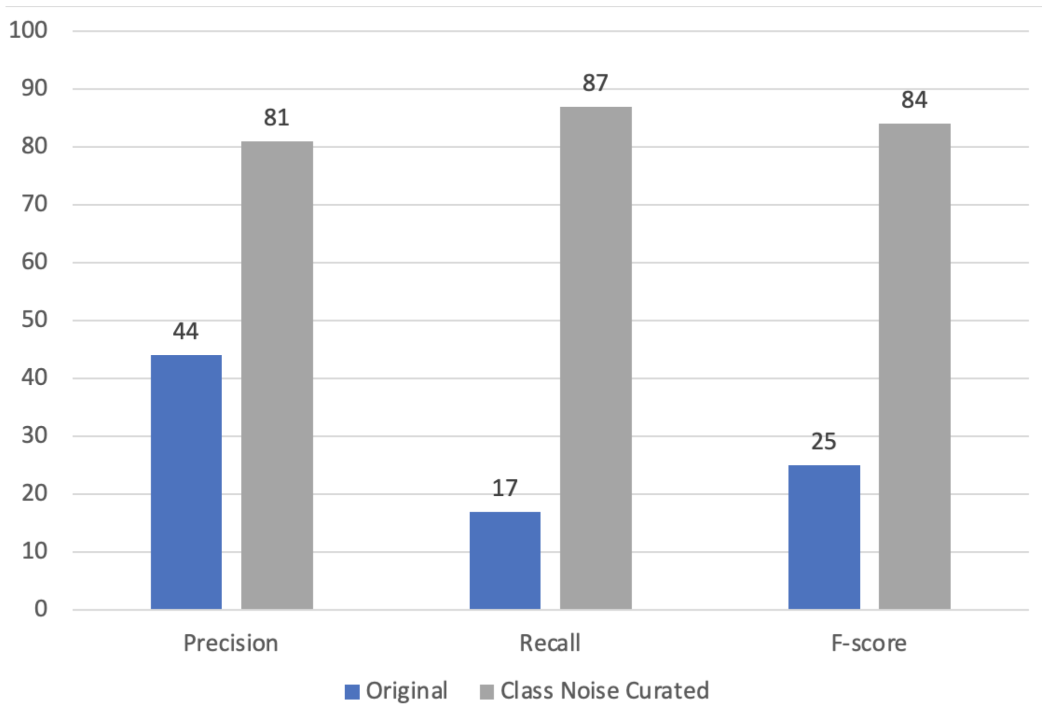


Figure 8: Learning Performance On the Original and the Class Noise cleaned Data Sets

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: Descriptive Statistics For Precision.

6.26.2. Since we have issues with normality in the samples, we decided to run

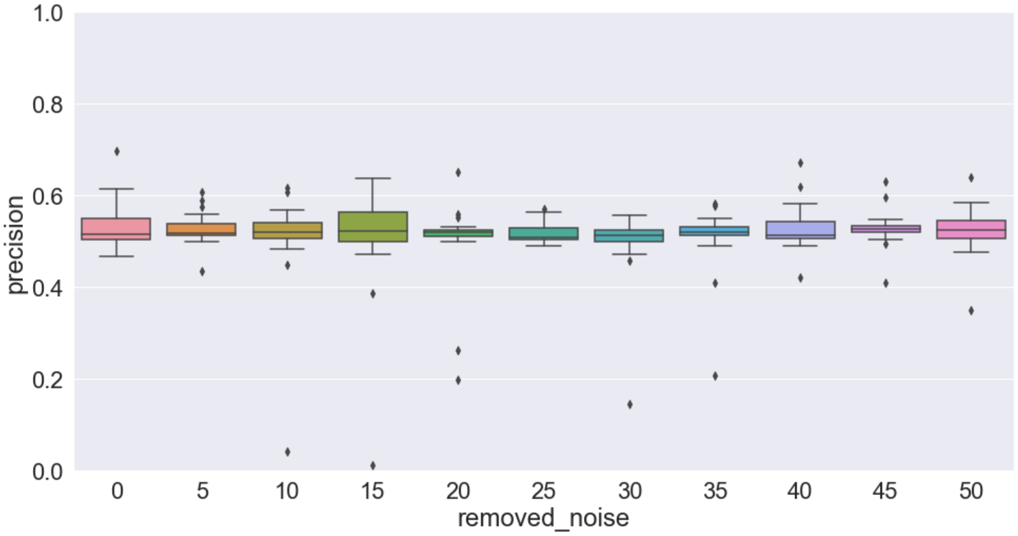


Figure 9: The Distribution of Precision Values Under the Treatment Levels

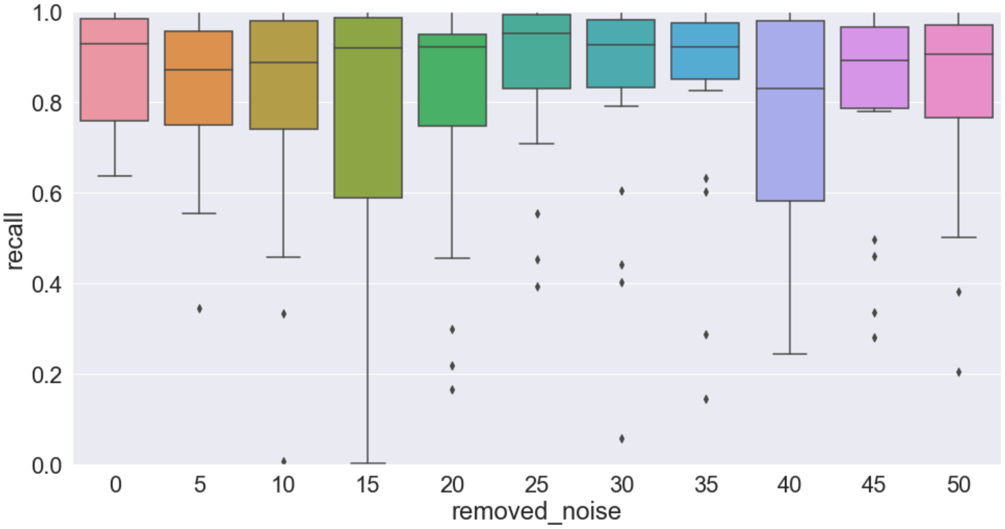


Figure 10: The Distribution of Recall Values Under the Treatment Levels

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 9 sum-

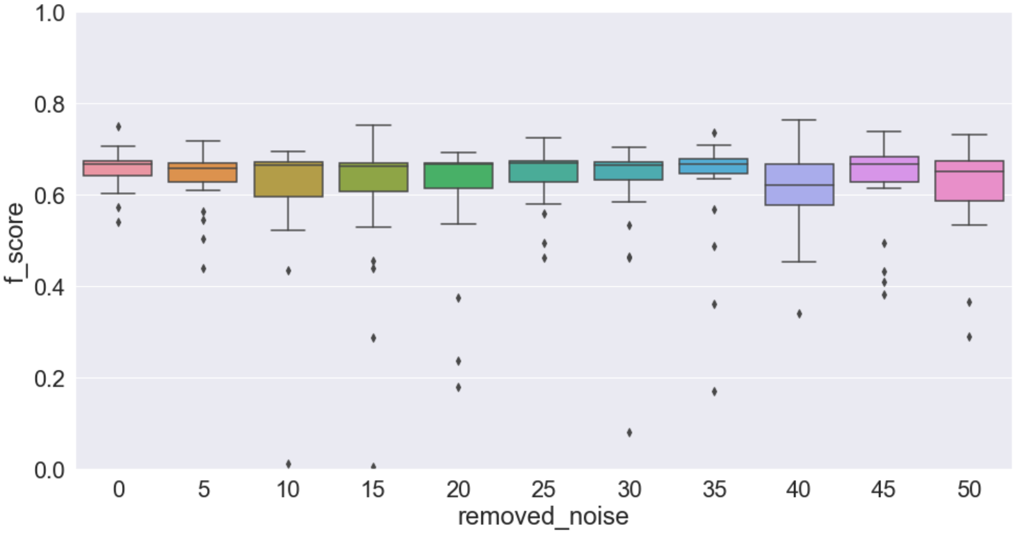


Figure 11: The Distribution of F-Score Values Under the Treatment Levels

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

Table 5: Descriptive Statistics For Recall.

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

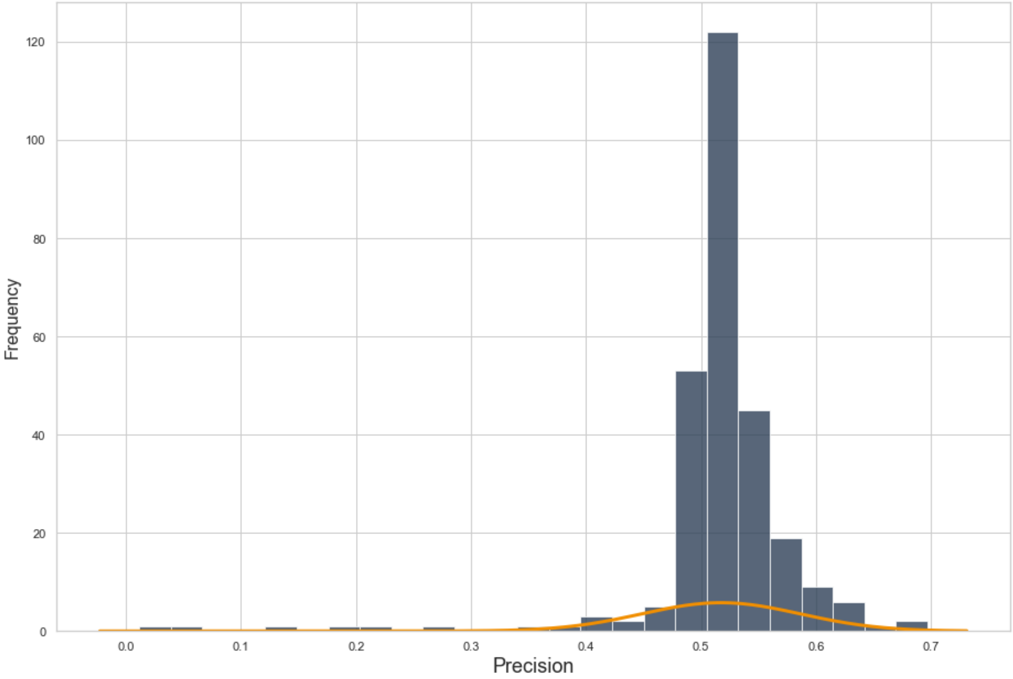


Figure 12: Frequency Plot For the Precision Scores

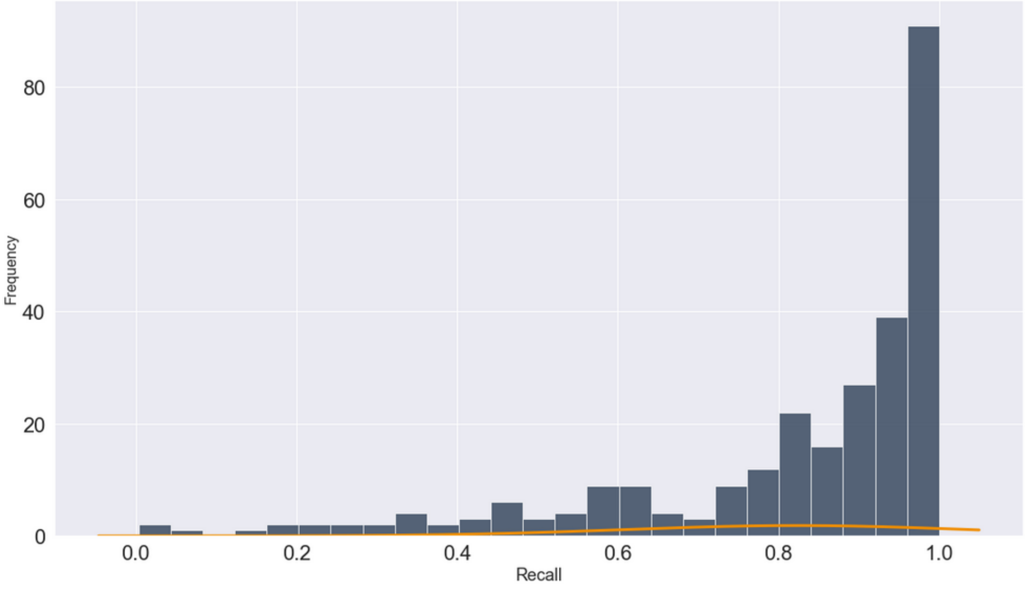


Figure 13: Frequency Plot For the Recall Scores

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

Table 6: Descriptive Statistics For F-Score.

	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
F-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

Table 7: The Shapiro-Wilk Results For Normality From 5% to 25% 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
F-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 8: The Shapiro-Wilk Results For Normality From 30% to 50% Treatment Levels.

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

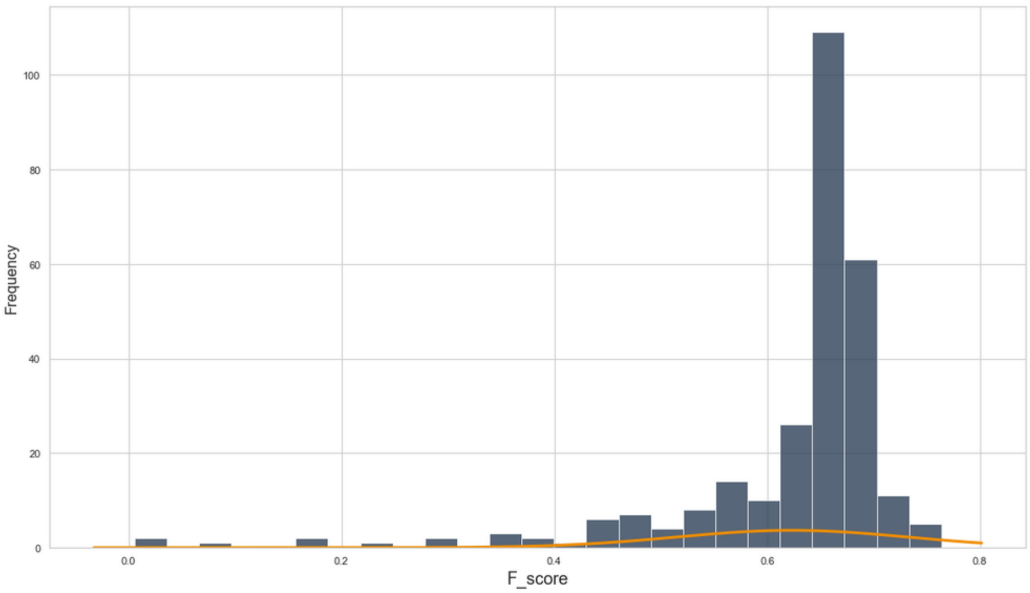


Figure 14: Frequency Plot For the F-Score Scores

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

Table 9: Statistical Results For the Comparison Between the Evaluation Measures Under All Treatment Levels.

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 6.1 and 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 f-score in Tables 4, 5, and 6, and Figure 8. Recall from Section 5.4.1 that the performance measures obtained at 0% treatment level (control group) are treated as the baseline measures. The remaining treatment levels are used to examine the effectiveness of handling attribute noise at different levels on the performance of the ML model used in MeBoTS.

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

- compared with the other two trials of training, using an uncleaned data-set for training provides the lowest learning performance.
- training a learner on a class-noise cleaned data would improve the performance of the learner by 70% recall, 37% precision, and 59% F-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 [29][15], 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.

8. Threats to Validity

When analyzing the threats to validity of our study, we followed the framework recommended by Wohlin et al.[40] 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, we can accept this threat, since the goal of this paper is to gain an initial understanding of the effect of attribute and class noise.

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 [20], 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[20]. 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[20] 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, $\hat{x}_i k$ 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 3.4) and the ranking of noisy observations were in line with the definition of attribute noise provided in the original publication [20].

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.

9. Conclusion and Future Work

In this paper, we set out to study the effect of class and attribute noise found in training data on the learning performance of an ML model for test case selection. We chose to study the effect of handling two types of noise (class and attribute) using a correction and an elimination based approaches. The results from this study suggest that handling class noise is important for improving the prediction of test case verdicts, whereas no similar conclusion about attribute noise could be drawn. As a consequence, testers who choose to use an ML-based approach for test case selection should consider handling class noise in the training data, but not necessarily attribute noise, for achieving a higher learning performance in the prediction of test case verdicts. 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.

References

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

[2] S. Kim, E. J. Whitehead Jr, Y. Zhang, Classifying software changes: Clean or buggy?, IEEE Transactions on Software Engineering 34 (2008) 181–196.

[3] M. Ochodek, R. Hebig, W. Meding, G. Frost, M. Staron, Recognizing lines of code violating company-specific coding guidelines using machine learning, Empirical Software Engineering 25 (2020) 220–265.

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

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

[6] Z. Cai, L. Lu, S. Qiu, An abstract syntax tree encoding method for cross-project defect prediction, IEEE Access 7 (2019) 170844–170853.

[7] T. B. Noor, 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, pp. 2–11.

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

[9] K. W. Al-Sabbagh, M. Staron, R. Hebig, 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), volume 2476, pp. 138–153.

[10] K. W. Al-Sabbagh, M. Staron, R. Hebig, 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, pp. 191–194.

[11] T. M. Khoshgoftaar, N. Seliya, 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, pp. 302–307.

[12] D. Guan, W. Yuan, Y.-K. Lee, S. Lee, Identifying mislabeled training data with the aid of unlabeled data, Applied Intelligence 35 (2011) 345–358.

[13] F. Muhlenbach, S. Lallich, D. A. Zighed, Identifying and handling mislabelled instances, Journal of Intelligent Information Systems 22 (2004) 89–109.

[14] T. Zimmermann, P. Weißgerber, Preprocessing cvs data for fine-grained analysis., in: MSR, volume 4, pp. 2–6.

[15] X. Zhu, X. Wu, Class noise vs. attribute noise: A quantitative study, Artificial intelligence review 22 (2004) 177–210.

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

[17] C. E. Brodley, M. A. Friedl, et al., Identifying and eliminating mislabeled training instances, in: Proceedings of the National Conference on Artificial Intelligence, pp. 799–805.

[18] T. M. Khoshgoftaar, J. Van Hulse, Identifying noise in an attribute of interest, in: Fourth International Conference on Machine Learning and Applications (ICMLA'05), IEEE, pp. 6–pp.

[19] K.-A. Yoon, D.-H. Bae, A pattern-based outlier detection method identifying abnormal attributes in software project data, Information and Software Technology 52 (2010) 137 – 151.

[20] J. D. Van Hulse, T. M. Khoshgoftaar, H. Huang, The pairwise attribute noise detection algorithm, Knowledge and Information Systems 11 (2007) 171–190.

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

[22] H. Hata, O. Mizuno, T. Kikuno, Fault-prone module detection using large-scale text features based on spam filtering, Empirical Software Engineering 15 (2010) 147–165.

[23] O. Mizuno, S. Ikami, S. Nakaichi, 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, p. 4.

[24] L. Aversano, L. Cerulo, 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, pp. 19–26.

[25] K. W. Al-Sabbagh, R. Hebig, 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, pp. 287–303.

[26] T. M. Khoshgoftaar, J. Van Hulse, Empirical case studies in attribute noise detection, IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews) 39 (2009) 379–388.

[27] C.-M. Teng, A comparison of noise handling techniques., in: FLAIRS Conference, pp. 269–273.

[28] J. R. Quinlan, Induction of decision trees, Machine learning 1 (1986) 81–106.

[29] C. E. Brodley, M. A. Friedl, Identifying mislabeled training data, Journal of artificial intelligence research 11 (1999) 131–167.

[30] M. Ochodek, M. Staron, D. Bargowski, W. Meding, 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, pp. 14–20.

[31] R. Moser, W. Pedrycz, 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, pp. 181–190.

[32] S. Amasaki, Y. Takagi, O. Mizuno, 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, pp. 215–226.

[33] J. Deng, L. Lu, S. Qiu, Y. Ou, A suitable ast node granularity and multi-kernel transfer convolutional neural network for cross-project defect prediction, IEEE Access 8 (2020) 66647–66661.

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

[35] A. Hamou-Lhadj, 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, pp. 42–55.

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

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

[38] V. Ganganwar, An overview of classification algorithms for imbalanced datasets, International Journal of Emerging Technology and Advanced Engineering 2 (2012) 42–47.

[39] 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, E. Duchesnay, Scikit-learn: Machine learning in Python, Journal of Machine Learning Research 12 (2011) 2825–2830.

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

10. Appendices

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