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

Citation for the original published paper (version of record):
Al Sabbagh, K., Staron, M., Hebig, R. (2020). The Effect of Class Noise on Continuous Test Case Selection: A Controlled Experiment on
Industrial Data. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 12562 LNCS: 287-303.
http://dx.doi.org/10.1007/978-3-030-64148-1_18

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

Khaled Walid Al-Sabbagh$^{1[0000-0003-2571-5099]}$, Regina Hebig$^{1[0000-0002-1459-2081]}$, and Miroslaw Staron$^{1[0000-0002-9052-0864]}$

Chalmers | University of Gothenburg, Computer Science and Engineering Department, Gothenburg, Sweden
{khaled.al-sabbagh, miroslaw.staron, regina.hebig}@gu.se

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, F-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, F-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%.

Keywords: controlled experiment, class noise, test case selection, continuous integration

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 [2][8][10]. 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 [8][3][2].

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 [9][20][6]. Attribute noise refers to corruptions in the feature values of instances in a data-set. Examples include: missing and incomplete feature values [16]. 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) [21]. 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?*

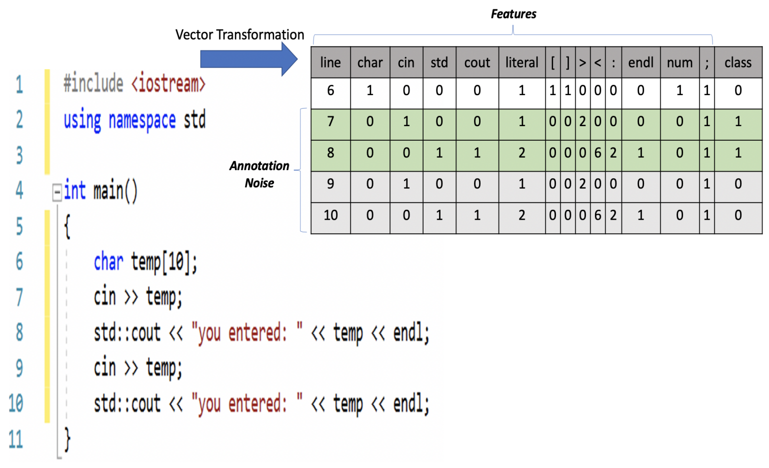
2 Definition and Example of class Noise in Source Code

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 [7] or entropy measurements [17]) and therefore we count all such entries as contradictory. As an illustration of the problem, in the domain of TCS, Figure 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 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 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:

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



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

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

Fig. 1. Class Noise in Code Base.

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 Related Work

Several studies have been made to identify the effect of class noise on the learning of ML models in several domains[12][19][1]. 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.

3.1 The Impact of Noise on Classification Performances

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.[12] 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 [19] 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 f-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. [14] 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 [1] 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.

3.2 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 [2].

A previous work on TCS [2] 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. [8] 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. [11] 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. [10] 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.

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.

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 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 [13] 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 5.3)

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 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

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

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.

4.3 Dependent Variables

The dependent variables are four evaluation measures used for the performance of an ML classifier – Precision, Recall, F-score, and Matthews Correlation Coefficient (MCC)[4]. 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 F-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.

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 F-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.

4.5 Data Analysis Methods

The experimental data were analyzed using the scikit learn library with Python [15]. 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 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.

5 Experiment Operations

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

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 5.2). The classification performance in the treatment groups will then be compared to that in the control group (see Section 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 [5]. Therefore, we decided to start from our industrial code changes data-set (See Section 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 [15] 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 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 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, f-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: 3679 entries labeled as failed and 4280 entries as pass.
– Entries with no duplicates in the data-set: 1 entry labeled as failed and 1370 entries as passed.

5.2 Class Noise Generation

To generate class noise into the experimental subjects, we followed the definition of noise introduced in section 2 by carrying out the following two-steps procedure:

1. Given a noise ratio Nr, we randomly pick a portion of Nr from the population of duplicate vectors within each class in the training and validation subjects.
2. We re-label half of the label values of duplicate entries selected in step 1 to the opposite class to generate Nr noise ratio. In situations where the number of duplicate entries in Nr are uneven, we re-label half of the selected Nr portion minus one entry.

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

A common issue in supervised ML is that the arithmetic classification accuracy becomes biased toward the majority class in the training data-set, which might lead to the extraction of poor conclusions. This effect might be magnified if noise was added without checking the balance of classes after generating noise. In this experiment, due to the large computational cost required to check the distribution of classes across 90 trials, we only checked the distribution under 10% noise ratio. Figure 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

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

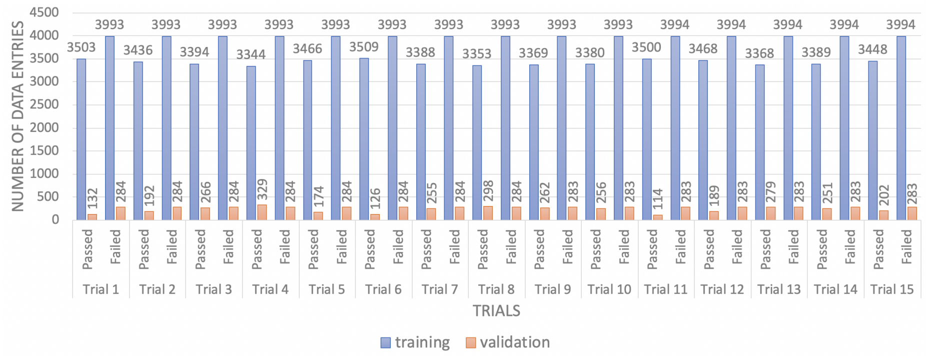


Fig. 2. The Distribution of The Binary Classes After Generating Noise at 10% Ratio.

the training and validation sets. The Figure shows a fairly balanced distribution in the training subjects with an average of 3421 entries in the passed class and 3993 entries in the failed class.

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.

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.

6.1 Descriptive Statistics

The descriptive statistics are presented in Tables 2, 3, 4, and 5 individually for each dependent variable. The values for Precision (Table 2), Recall (Table 3), F-score (Table 4), and MCC (Table 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, f-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 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 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. Descriptive Stats For Recall.

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

Table 4. Descriptive Stats For F-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 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

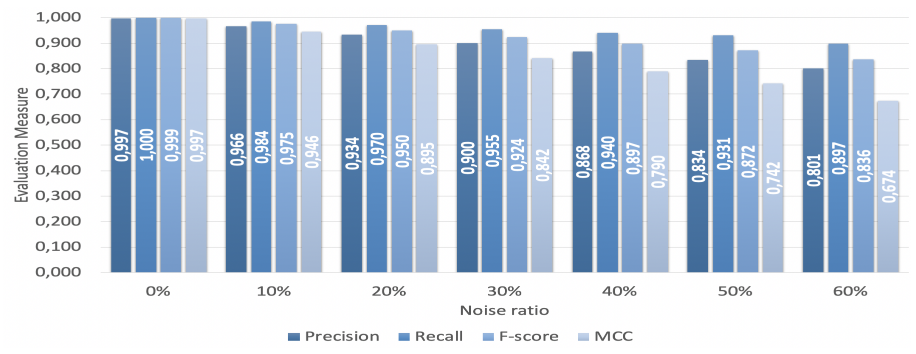


Fig. 3. Mean Distribution of the Evaluation Measures.

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-

Wilk test results were statistically significant for all the evaluation measures in the majority of the noise ratios. Table 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 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 7 summarizes the statistical comparison results, indicating a significant difference in Precision, F-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 F-score and MCC (F-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 6. Statistical Results For Normality.

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

Table 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
f-score	p<0.005	Statistics=54.172
mcc	p<0.005	Statistics=53.398

The Mann–Whitney U test with Precision, F-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 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 8. The Comparison Results From Mann-Whitney Test

	10%	20%	30%	40%	50%	60%
Precision	Stat=7.5, p<0.005	Stat=0.000, p<0.005	Stat=0.000, p<0.005	Stat=0.000, p<0.005	Stat=0.000, p<0.005	Stat=0.000, p<0.005
Recall	Stat=45, p=0.184	Stat=40.000, p=0.084	Stat=40.000, p=0.084	Stat=35.000, p<0.005	Stat=30.000, p=0.017	Stat=25, p=0.007
F-Score	Stat=7.5, p<0.005	Stat=0.000, p<0.005	Stat=0.000, p<0.005	Stat=0.000, p<0.005	Stat=0.000, p<0.005	Stat=0.000, p<0.005
MCC	Stat=7.5, p<0.005	Stat=0.000, p<0.005	Stat=0.000, p<0.005	Stat=0.000, p<0.005	Stat=0.000, p<0.005	Stat=0.000, p<0.005

7 Threats to Validity

When analyzing the validity of our study, we used the framework recommended by Wohlin et al. [18]. 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.

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 data-set. 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.

References

1. Abellán, J., Masegosa, A.R.: Bagging decision trees on data sets with classification noise. In: International Symposium on Foundations of Information and Knowledge Systems. pp. 248–265. Springer (2010)
2. Al-Sabbagh, K.W., Staron, M., Hebig, R., Meding, W.: 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, pp. 138–153 (2019)
3. Aversano, L., Cerulo, L., Del Grosso, C.: 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. pp. 19–26. ACM (2007)
4. Boughorbel, S., Jarray, F., El-Anbari, M.: Optimal classifier for imbalanced data using matthews correlation coefficient metric. PloS one **12**(6) (2017)
5. Frénay, B., Verleysen, M.: Classification in the presence of label noise: a survey. IEEE transactions on neural networks and learning systems **25**(5), 845–869 (2013)
6. Gamberger, D., Lavrac, N., Dzeroski, S.: Noise detection and elimination in data preprocessing: experiments in medical domains. Applied Artificial Intelligence **14**(2), 205–223 (2000)

7. Guan, D., Yuan, W., Shen, L.: Class noise detection by multiple voting. In: 2013 Ninth International Conference on Natural Computation (ICNC). pp. 906–911. IEEE (2013)
8. Hata, H., Mizuno, O., Kikuno, T.: Fault-prone module detection using large-scale text features based on spam filtering. Empirical Software Engineering **15**(2), 147–165 (2010)
9. John, G.H.: Robust decision trees: Removing outliers from databases. In: KDD. vol. 95, pp. 174–179 (1995)
10. Kim, S., Whitehead Jr, E.J., Zhang, Y.: Classifying software changes: Clean or buggy? IEEE Transactions on Software Engineering **34**(2), 181–196 (2008)
11. Mizuno, O., Ikami, S., Nakaichi, S., Kikuno, T.: Spam filter based approach for finding fault-prone software modules. In: Proceedings of the Fourth International Workshop on Mining Software Repositories. p. 4. IEEE Computer Society (2007)
12. Nettleton, D.F., Orriols-Puig, A., Fornells, A.: A study of the effect of different types of noise on the precision of supervised learning techniques. Artificial intelligence review **33**(4), 275–306 (2010)
13. Ochodek, M., Staron, M., Bargowski, D., Meding, W., Hebig, R.: Using machine learning to design a flexible loc counter. In: 2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE). pp. 14–20. IEEE (2017)
14. Pechenizkiy, M., Tsymbal, A., Puuronen, S., Pechenizkiy, O.: Class noise and supervised learning in medical domains: The effect of feature extraction. In: 19th IEEE symposium on computer-based medical systems (CBMS'06). pp. 708–713. IEEE (2006)
15. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research **12**, 2825–2830 (2011)
16. Sáez, J.A., Luengo, J., Herrera, F.: Evaluating the classifier behavior with noisy data considering performance and robustness: The equalized loss of accuracy measure. Neurocomputing **176**, 26–35 (2016)
17. Sluban, B., Lavrač, N.: Relating ensemble diversity and performance: A study in class noise detection. Neurocomputing **160**, 120–131 (2015)
18. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in software engineering. Springer Science & Business Media (2012)
19. Zhang, J., Yang, Y.: 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. pp. 190–197 (2003)
20. Zhao, Q., Nishida, T.: Using qualitative hypotheses to identify inaccurate data. Journal of Artificial Intelligence Research **3**, 119–145 (1995)
21. Zhu, X., Wu, X.: Class noise vs. attribute noise: A quantitative study. Artificial intelligence review **22**(3), 177–210 (2004)