The Impact of Class Noise-handling on the Effectiveness of Machine Learning-based Methods for Build Outcome and Code Change Request

The Impact of Class Noise-handling on the Effectiveness of Machine Learning-based Methods for Build Outcome and Code Change Request Predictions

KHALED AL-SABBAGH and MIROSLAW STARON, University of Gothenburg, Goteborg, Sweden
REGINA HEBIG, University of Rostock, Rostock, Germany

Machine learning-based methods are increasingly used to optimize build processes and accelerate the integration of software code. These methods leverage large volumes of historical code changes to train models on predicting and preventing issues in the codebase that could delay code integrations and features delivery to end-users. The objective of this study is to examine the impact of handling class noise present in software code changes collected from Continuous Integration (CI) systems on the predictive performance of machine learning models for predicting the execution outcome of CI builds and negative code reviews. In this study, we conduct a series of computational experiments using data from 110 Java open-source projects, examining the effectiveness of two removal-based statistical techniques - Majority Filter (MF) and Consensus Filter (CF) - and two corrective techniques - Domain Knowledge-based (DB) and CleanLab. Our results show that removal-based techniques significantly improve model predictive performance in both build outcome and negative code review prediction tasks. For build outcome prediction, applying MF increased the F1-score from 82% to 97%, and MCC from 0.13 to 0.58. In negative code review predictions, MF improved the F1-score from 17% to 53%, and MCC from −0.03 to 0.57. The DB technique was effective primarily in the context of code review comments but less so for build outcome predictions. While CleanLab yielded more consistent predictions, its overall impact on model performance was more moderate compared to removal-based techniques. Additionally, our findings show that hyperparameter tuning, applied independently or in combination with CleanLab, can further improve model performance; however, these gains did not surpass those achieved by removal-based techniques alone. We conclude that applying removal-based techniques to the training data of code changes is necessary to improve the prediction of build outcomes and negative code review comments.

CCS Concepts: • **Software and its engineering → Software verification and validation**;

Additional Key Words and Phrases: Machine learning, Build outcomes, negative code review comments, and Class noise-handling

ACM Reference format:
Khaled Al-Sabbagh, Miroslaw Staron, and Regina Hebig. 2026. The Impact of Class Noise-handling on the Effectiveness of Machine Learning-based Methods for Build Outcome and Code Change Request Predictions. *ACM Trans. Softw. Eng. Methodol.* 35, 6, Article 147 (May 2026), 66 pages.
https://doi.org/10.1145/3764864

Authors' Contact Information: Khaled Al-Sabbagh (corresponding author), University of Gothenburg, Goteborg, Sweden; e-mail: khaledwalidsabbagh@gmail.com; Miroslaw Staron, University of Gothenburg, Goteborg, Sweden; e-mail: miroslaw@chalmers.se; Regina Hebig, University of Rostock, Rostock, Germany; e-mail: regina.hebig@uni-rostock.de.

ACM 1557-7392/2026/5-ART147
https://doi.org/10.1145/3764864

1 Introduction

Software engineering companies are under increasing pressure to deliver high-quality software products to the end-users as quickly as possible [30]. To meet these demands, companies are adopting the practice of **Continuous Integration (CI)**, which promotes for frequent integration and testing of code changes [14]. This practice has become an integral part of modern software development processes as it offers various advantages, such as streamlining workflows, fostering collaboration between team members, and improving software quality.

Despite these benefits that CI offers to companies, it poses the challenge of minimizing the feedback latency between CI and software engineers without compromising fault detection capability. The growing complexity of features and the lengthy compilation time that CI takes underline the need for tools that can expedite the feedback time to software engineers and effectively pinpoint faults in the code. These tools aim to promptly identify and report code changes that are likely to contain faults to software engineers, ensuring rapid detection and fixing of faults.

Machine Learning (ML)-based methods have proven effective in accelerating code integration by predicting fault-prone software modules [23, 39]. By analyzing code commit histories and build execution outcomes, these methods can identify patterns in code changes that may lead to build failures [2]. However, recent studies have shown that code commits data come with large amounts of inaccurate class values, known as class noise, assigned to lines of code can impede the predictive performance of ML models [21]. In the context of CI, these inaccurately labeled lines of code can originate from several factors, such as flaky test cases or flawed data collection methods.

To address the problem of class noise, researchers have proposed several techniques that can be divided into three categories: (1) tolerance, (2) removal, and (3) correction. The tolerance-based techniques focus on designing ML models that are robust and capable of tolerating a certain level of noise in the data [62]. The removal-based techniques seek to identify entries with class noise and then remove them from the dataset. The main advantage of using such techniques is that they retain cleaner entries of the data and discard unclean ones. However, using removal-based techniques can lead to losing valuable information that can be used by the model to learn important patterns. Finally, techniques in the correction category seek to correct mislabeled entries by replacing their values with ones that are more appropriate. These techniques are particularly appealing when the training data is small since no removal of data entries is required. However, by correcting mislabeled entries we introduce a risk of bias toward one of the classes [32].

Several **Software Engineering (SE)** researchers have investigated the use of different noise-handling techniques to improve the detection of fault-prone software modules. For example, Khoshgoftaar and Rebours [26] conducted a case study to investigate the effectiveness of two removal-based techniques in improving the identification of fault-prone software modules. The evaluation was performed using a dataset of 10,883 software modules. The results showed that the models' performances improve the most when using a conservative technique where fewer entries in the data are removed. In a related study, Hulse et al. [54] investigated a correction-based approach using Bayesian multiple imputation to address class noise. Working with a dataset of 282 software program modules, they reported that approximately 60% of mislabeled entries were successfully corrected using the technique. Additionally, Khoshgoftaar et al. [27] proposed a removal-based method combining statistical modeling and Boolean rules to detect noisy entries. Their evaluation, conducted across seven software programs from the NASA dataset, demonstrated that the approach successfully identified all known noisy entries in six out of seven projects.

While most existing studies in the literature demonstrate that noise-handling techniques can significantly improve model predictive performance in SE tasks, these studies primarily focus on examining statistical-based methods without considering domain-specific factors that could

influence noise patterns. Moreover, few research has examined the effectiveness of corrective-based techniques in the context of CI. To address these gaps, we extend prior research by evaluating both removal-based and corrective-based noise-handling techniques in the context of CI. Specifically, we assess two widely used removal-based methods—**Majority Filter (MF)** and **Consensus Filter (CF)**—alongside two corrective approaches: a domain knowledge-driven technique and CleanLab, a data-driven framework that automatically detects and corrects mislabeled instances [35]. Our goal is to empirically examine and compare the impact of these techniques that span removal and corrective strategies on the performance of a ML-based method for predicting fault-prone code changes using large-scale datasets of code commit histories. By doing so, we aim to provide DevOps practitioners and software engineers with actionable insights into which class noise-handling strategies are most effective for improving the prediction of fault-prone changes within CI pipelines.

To achieve this goal, we design and implement two computationally controlled experiments in which we examine the impact of two statistical-based removal techniques and two corrective-based techniques for class noise-handling in the context of CI. In the first experiment, we examine the impact of these techniques on the prediction of build execution outcomes, i.e., whether a build will fail or pass. We perform the evaluation using a large dataset of 110 Java open source projects and pose the following **Research Question (RQ)**:

RQ1: What is the impact of applying class noise-handling techniques on predicting the outcome of builds in CI?

The second experiment examines how each of the four class noise-handling technique impacts the prediction of negative code review comments made by reviewers during the code review process. We work with a sample of 5,066 lines of code and their corresponding code comments from two open source projects. These lines of code are manually labeled based on the sentiment expressed by reviewers regarding the readiness of the code for integration or the need for changes. Consequently, we pose the following RQ.

RQ2: What is the impact of applying class noise-handling techniques on predicting negative code review comments?

Since ML techniques inherently rely on numerical vectors as input for training, the selection of the feature extraction algorithm becomes a crucial factor in influencing the predictive performance of the trained model.

Consequently, in this study, we extend our analysis of the impact of noise-handling by examining the effectiveness of two feature extraction techniques: **Bag-of-words (BoW)** and CodeBERT [15]. We selected these two techniques due to their popularity in the software engineering literature and their demonstrated potential in improving the prediction of software engineering tasks [6, 37, 61]. To evaluate their impact, we conducted experiments using a sample of nine projects for build outcome prediction and two projects for code change request prediction. Accordingly, we pose the following RQ:

RQ3: To what extent do BoW and CodeBERT feature extraction techniques impact the performance of noise-handling techniques?

In addition to feature extraction, existing literature shows that the performance of ML classifiers can be significantly influenced by hyperparameter tuning [50]. Hyperparameters control key aspects of the model's behavior, such as the number of decision trees in a **Random Forest (RF)** classifier, and parameter tuning can substantially improve predictive accuracy. Hyperparameter tuning is typically performed using either grid search or random search [8]. The grid search systematically explores all possible combinations of specified hyperparameter values, while the random search

samples a subset of possible combinations randomly. Although random search is often more efficient for high-dimensional search spaces, grid search offers a more exhaustive evaluation, increasing the likelihood of identifying optimal hyperparameter settings [40].

This study further extends the analysis by examining the impact of hyperparameter tuning on predicting build execution outcomes in the presence of class noise. Understanding how tuning affects model performance is particularly critical in noisy datasets, where inappropriate parameter settings can exacerbate model degradation. Therefore, we pose the following RQ:

RQ4: To what extent does model hyperparameter tuning impact the prediction of build outcomes before and after handling class noise?

The evaluation of the impact of the noise-handling techniques and feature extraction algorithms is performed by measuring and comparing the predictive performance of a RF model in terms of Precision, Recall, F1, and MCC for build outcome and code change request predictions. Our results from the first experiment suggest that applying the removal-based techniques would consistently improve the predictive performance of the model in terms of Precision, Recall, F1, and MCC. However, our results suggest that applying any of the three class noise-handling techniques will consistently improve the predictive performance of the model, albeit to a varying degree.

Our results show that while all noise-handling techniques improve model performance to varying degrees, hyperparameter tuning is most effective when applied before noise-handling or in combination with CleanLab.

This article is structured as follows: Section 2 discusses previous research studies that are related to the study presented in this article. Section 3 presents background information, covering core concepts and a formal definition of how class noise is measured in code commits data. Section 4 describes the research design. Section 5 presents the evaluation results of the impact of class noise-handling techniques on the performance of the ML-based method for build outcome and negative review comment predictions. Section 6 answers the RQs and discusses general observations. Section 7 addresses the threats to validity of this study. Finally, Section 8 presents the findings and concludes the article.

2 Related Work

In this section, we begin by presenting related work on noise-handling techniques. Then, existing studies on the impact of noise-handling on the performance of predictive models for SE tasks are discussed. Finally, we discuss previous studies that proposed build outcome and code change request approaches and how these work are subject to class noises.

Since our study considers noise-handling techniques that belong to the removal and correction categories, we specifically focus on discussing techniques that fall under these two categories.

2.1 Removal and Correction-based Techniques for Class Noise-handling

Removal-based Techniques. Brodley and Friedl [9] proposed a method for identifying and handling noise in training data using three or five learning algorithms as filters. These filters employ majority voting or CF mechanisms to identify potentially noisy instances. The effectiveness of their approach was assessed using five different datasets. The evaluation results indicated that, when dealing with noise levels of 20% and higher, the MF exhibited slightly better prediction accuracy compared to the CF. However, for noise levels below 20%, both filters demonstrated similar accuracy scores ranging from 79% to 82%.

Guan et al. [20] proposed a variant of the majority and CFs that incorporated a semi-supervised classification step to aid in predicting class values for unlabeled instances. The technique was tested on 20 benchmark datasets and evaluated by measuring the classification error rate. The comparisons demonstrated that the technique improved the classification performance of ML models across

four different noise ratios. Specifically, when the noise ratio was 10%, the technique reduced the classification error rate by 4.5%, while at a noise ratio of 40%, the improvement significantly increased to 25.6%.

Wilson and Martinez [56] introduced an instance-based technique called DROP3, which aims to remove noisy entries from the data using a distance function. The technique determines the proximity of each input vector in a subset of the training data to the entire dataset and consequently determine which input vector has noise and requires removal. To evaluate its effectiveness, DROP3 was compared with 10 other techniques across 31 classification tasks. The comparison results based on average accuracy demonstrated that DROP3 ranked 5th in terms of its improvement effect on accuracy, with an average of 81.14%.

Chen et al. [10] proposed a method that combines class noise-handling and ensemble imbalance learning methods to improve software defect prediction. The method uses the **Neighborhood Cleaning Learning (NCL)** rule to locate and remove instances that have contradictory nearest neighbors in unbalanced datasets. The evaluation of the method was done using nine datasets from the PROMISE repository, where the authors applied and compared the effectiveness of their class noise-handling method with two existing class noise-handling methods to each dataset. The evaluation showed that using the proposed method outperformed the other two methods in terms of Geometric Mean and **Area under the ROC Curve (AUC)** with a large effect size. Particularly, the proposed method achieved a total average value of 74.9% in Geometric Mean, and 82.1% in AUC.

Gong et al. [18] examined the effect of three class noise removal-based techniques: (1) the NCL [10], (2) the k-means Clustering Cleaning Algorithm [51]—a clustering-based noise detection algorithm, and (3) an **Improved k-means Clustering Cleaning Algorithm (IKMCCA)** on the performance of a model for software defect prediction. Using 28 projects from the NASA, AEEEM, ReLink, SOFTLAB, and MORPH datasets, the authors evaluated the impact of the three class noise-handling algorithms on the prediction performance of existing within-project defect prediction models. The results showed that using the three class noise-handling algorithms can result in better medium values of Recall and AUC for Naive-Bayesian, RF, Support Vector Machine, **K-nearest Neighbors (KNN)**, and **Linear Regression (LR)** classifiers. Particularly, the highest median Recall was observed when applying the IKMCCA and when using the KNN algorithm (Recall and AUC at 75%).

In another study, Gong et al. [19] proposed a method that analyzes the class distribution in the local neighborhood of a given instance for class noise identification. Using 230 datasets and two baseline methods, the authors examined (1) the impact of different levels of class noise on the performance of a model for defect prediction, and (2) the impact of their method on the performance of seven software defect prediction models. The results showed that a dataset with class noise ratio above 12.5% had a significant effect on the stability of the performance of defect prediction models. In addition, applying the examined class noise-handling techniques could improve the performance of defect prediction models when the training dataset has over 12.5% class noise ratios. Particularly, applying the proposed class noise-handling method on all of the datasets improved the classification performance of seven out of seven studied models. The highest improvement in classification, as measured in AUC, was achieved when training a RF model—AUC improved by 4.5% on average across all the analyzed datasets.

Correction-based Techniques. Muhlenbach et al. [34] introduced an algorithm that allows users to filter and polish noisy instances. The algorithm utilizes neighborhood graphs and cut edge weight algorithms to identify noisy suspects. An instance is considered noisy when its class value is different than one of the examples belonging to its geometrical neighborhood. Once noisy suspects are identified, they can either be removed or removed and relabeled. Muhlenbach et al. evaluated their algorithm using datasets extracted from 10 benchmark ML repositories. The findings suggest

that removing noisy suspects from the training data produces better results in 9 out of 10 datasets compared to removing and relabeling noisy suspects at a noise level between 4% and 10%.

Teng [52] introduced a noise correction technique that exploits the interdependence relationship between the attribute and class values to identify and correct inaccurate values. The idea is to use an ensemble classifier to predict noisy suspects in the data. Some entries that were incorrectly classified by the ensemble (using a voting scheme) are tagged as noisy and then corrected. The author evaluated the effectiveness of her technique by measuring the prediction accuracy of Decision Trees before and after correction was applied. The results showed that at an intermediate noise level (20%–30%), the improvements in accuracy were significant. However, at 40% noise level, the improvements in accuracy were inconsistent.

Recently, Northcutt et al. [35] introduced a data-centric approach called confidence learning [29] that leverages probabilistic thresholds to estimate class noise in the data. The approach was evaluated for effectiveness in identifying label errors (i.e., class noise), using the CIFAR, MNIST, and ImageNet benchmarks. The results showed that Confident Learning accurately detected a large proportion of mislabeled examples with 56%–74% Precision, 90%–97% Recall, and 71%–79% F1-score. Furthermore, models retrained on corrected datasets demonstrated substantial gains in both robustness and predictive performance, highlighting the practical value of label noise estimation and correction.

The majority of these statistical-based approaches for class noise-handling require leveraging ML classifiers to identify noisy patterns in the data, which can be computationally expensive. Consequently, noise can only be identified if it has been introduced systematically, i.e., in cases where random labeling errors are present, these statistical approaches may fall short in accurately identifying noise.

On the other hand, the DB approach stands out as a lightweight approach, since it doesn't require training models to identify noisy patterns in the data. In addition, DB inherits the advantage of being domain-specific within the context of CI, i.e., an entry (line of code) that has been seen as part of a positive class (passing build or approved code changes) is not likely to trigger build failure or code change request. Therefore, the effectiveness of DB is not determined by the nature of labeling errors—systematic or random.

2.2 Class Noise-handling in Software Engineering Contexts

The most related studies to our work concern the analysis of the impact of class noise-handling on the predictive performance of ML models for SE tasks. This section discusses studies that examine the impact of class noise-handling techniques on the predictive performance of ML models in SE contexts.

Table 1 summarizes previous research studies that examined the effect of class noise-handling techniques on the predictive performance of ML models in SE contexts.

Kim et al. [28] proposed the Closest List Noise Identification technique for removing mislabeled entries in software defect datasets. The technique uses Euclidean Distance for measuring similarities between entries in the training data and comparing their class values. If the percentage of entries that have different class values, relative to an entry, is above a predefined threshold, then that entry is treated as noisy and, thus, removed from the dataset. The technique was evaluated on datasets from the Eclipse 3.4 SWT and Debug open source projects. The prediction results attained by an SVM model showed that F1 improves from 34% to 71% when the noise level is exactly at 30%. On the other hand, F1 decreased for the same SVM model when the noise level was higher than 30% (F1 decreased from 35% to 24%).

Liebchen et al. [31] conducted an experiment to assess the performance of three class noise-handling techniques using a sample of 8,888 entries. The sample data describes completed software projects whose attributes capture projects' characteristics (e.g., project names and types). The

Table 1. Results Summary for Class Noise Experimentation in Software Engineering Research

Study	SE context	Datasets	Noise approach(s)	Results
Kim et al. [28]	Defects prediction	— Columba — Eclipse 3.4 — Scarab — SWT — Debug	— Removal	F1-score: 71%
Liebchen et al. [31]	Effort estimation	— EDS	— Removal — Correction	% of remaining noise: Removal: 11,24% Correction: 365%
Zhong et al. [60]	Defects prediction	— NASA: JM1 and KC2	— Removal	Recall: 77% in JM1– 91% in KC2
Seiffert et al. [47]	Defects prediction	— CCCS	— Tolerance	AUC: Ranged from 97% to 100%
Saidani et al. [46]	Build outcome	— contextlogger — SAX — future — solr-iso639-filter — GI — grammarviz2_src — parallec — candybar-library — steve — mtsar	— Correction	AUC: Ranged from 84% to 92%
Gallaba et al. [17]	Build outcome	— 1,276 projects	Removal	— 12% of passing builds are not actually passing — 9%–14% of builds are incorrectly labeled

examined class noise-handling techniques were (1) filtering, (2) robust filtering, and (3) filtering and polishing. The results of the experiment showed that using a robust filtering technique could eliminate 88% of noisy entries found in the initial dataset. Conversely, the filtering and polishing technique resulted in tripling the amount of noisy entries compared to the amount of noise found in the initial data.

Zhong et al. [60] proposed using an unsupervised learning technique followed by manual labeling by experts to deal with the problem of class noise and missing class labels in software-fault measurement data. The idea is based on the assumption that fault-prone software modules will have similar software measurements and, thus, can likely form clusters. The authors classified their technique as a removal-based approach since it offers experts the ability to decide whether all grouped fault-prone software modules should be labeled as such or not, and accordingly decide whether to keep or remove them from the data. The evaluation of the technique was done by comparing the mislabeled software modules by SE experts with software modules tagged by another removal-based technique, described in [43]. The evaluation was done using two software projects (written in C++) from the NASA software projects. The results showed that their clustering technique achieved a noise Recall performance of 77% in the first project and 91% in the second project.

Seiffert et al. [47] conducted a series of experiments to investigate the impact of both class noise and class imbalance on the predictive performance of models built to predict fault-prone program modules. The authors investigated the robustness of 11 ML models in tolerating class noise when using seven data-sampling techniques. By seeding class noise into a dataset that contains 282 program modules written in Ada, the authors examined the impact of applying the seven sampling techniques in the presence of four different class noise levels on the performance of the 11 models. The average AUC showed that the Naive Bayes model performs better than all others at all noise levels and is relatively unaffected by the increase in noise ratio. Specifically, the average AUC of the Naive Bayes model ranged from 97% to 100% at the four seeded noise levels.

Saidani et al. [46] introduced a search-based method to predict CI Skip commits. This technique utilized the Strength-Pareto Evolutionary algorithm to distinguish between skipped and non-skipped commits. In their study, the authors evaluated their approach using both within-project and cross-project validations. The evaluation was done using a dataset comprising of 14,294 CI commits from 15 open source projects that used the Travis CI system. The authors applied the class noise-handling approach proposed by Northcut et al. [35] to the CI data before examining the effectiveness of their approach. The evaluation results showed that the proposed approach performed better than baseline methods with an average AUC scores of 92% and 84% in cross-validation and cross-project validations, respectively.

Gallaba et al. [17] proposed parsing the configuration files (.yml) of build jobs to identify build jobs that are marked as successful but actually contain breakages. Using information, such as the `allow_failure` property, would unveil builds that contain breakages and thereby identify and remove noisy instances in the data before building tools to solve software engineering problems. The analysis of 59,904 passing builds that belonged to 1,276 open source projects showed that 12% of these builds have an ignored failure. On the other hand, the authors identified noisy build records that were marked as failed, but were not addressed immediately by developers. These builds were assumed to be noisy since their breakage did not call for an immediate attention of developers to fix them. Of the 4,136 broken builds, 3,426 builds (83%) were not fixed in the immediately subsequent build. Overall, the authors concluded that one in every 7–11 builds (9%–14%) are incorrectly labeled.

From this brief overview, we observe that the evaluation of the techniques proposed by the majority of the reviewed studies was performed using relatively small sample datasets (1–15 software programs). Further, parsing the configuration files of CI pipeline allows the identification of noise from one source only. However, build jobs can be incorrectly labeled (noisy) due to several other sources, such as flaky tests and data collection methods.

Therefore, the study presented in this article adds to the literature by examining the effectiveness of four techniques using: (1) large real-world datasets of historical code commits in two unexplored SE contexts (i.e., build outcome and negative code review comments), and (2) a reliable performance measure—MCC—which reveals more truth about the bias introduced by the noise-handling techniques [11].

3 Background

This section provides the definition of class noise that we used in this study. It also explains how the code review process in Gerrit is carried out, and describes the class noise-handling techniques that we examine for effectiveness in this study.

3.1 Definition and Example of Class Noise

In this study, we define class noise as *the ratio of contradictory entries in each class to the total number of entries in the data.* `contradictory entries` are entries that have the same vector representation and are labeled with different class values. Based on this definition, we use the following formula

to measure the ratio of class noise in the data:

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

Since a contradictory entry can only be among two or more entries, thus their total count in the data is calculated by summing up the number of the same entries that appear with one or more different class value(s). For example, a dataset containing six of the same entries with five that are labeled with `True` and one labeled that is labeled with `False` has six contradictory entries. It is not possible to define a general rule to identify which class label is correct based on the number of entries [31]. For example, noise sources might systematically tend to introduce False "False" class values. Since we do not know exactly which class value should be used in a specific context, we cannot simply relabel any entry, as suggested by the currently used solutions (e.g., using entropy measurements [49]), and therefore we count all such entries as contradictory.

3.2 Class Noise Example in Build Data

Automated builds within CI provide feedback to developers about changes in the codebase. A passing build indicates that code changes in a commit compile cleanly without issues, whereas a failing build indicates that there are issues in the commit that require fixing. Without a closer analysis of the nature of build failures, such as identifying the faulty lines of code in a commit, practitioners may inadvertently develop automated tools that use mislabel lines within code commits. This mislabeling often occurs when the overall build outcome (pass/fail) is considered as the ground-truth for labeling each line of code in a historical commit, leading to inaccurate labeling.

Further, the mislabeling can be attributed to the representation of input source code by the feature extraction algorithm. For example, using the BoW algorithm to represent lines of code may lead to generating identical feature vectors for lines that are syntactically similar but semantically different. Figure 1 exemplifies a scenario from the Hydra[1] source-code data where class noise can be introduced as a result of using a uni-gram BoW model. The figure depicts two commits submitted to the development repository. Commit 1 compiled successfully, passing all checks by the build jobs and resulting in a successful build. Commit 2 led to a failed build due to issues in the code. Applying a uni-gram BoW model to represent each line of code in the two commits led to the tabular representations presented on the right-hand side of the figure. In the tabular representation of commit 1, each line of code is labeled with class value 1, denoting a passing build, while in commit 2, each line is labeled with class value 0, indicating a failing build. By observing the feature vectors and their corresponding class values in both tabular representations, we notice that the following lines of code are duplicates and hold different class values, i.e., noisy:

—line 1 and line 10,
—line 2 and line 11, and
—line 4 and line 13.

3.3 Code Review Process in Gerrit

Over the last few decades, the process of code review has started to become more prevalent in the software engineering community. Big companies like Google started adopting lightweight tools to accelerate the review process [45]. A reviewer process typically enacts four sequential steps: (1) the author of a commit submits it to the code review tool, (2) the reviewers review the changes introduced in the submitted commit, (3) the reviewers can then either initiate a discussion thread on specific lines/blocks in the committed code or approve the changes, (4) the author of the commit

[1] https://github.com/cardano-foundation/hydra-java.

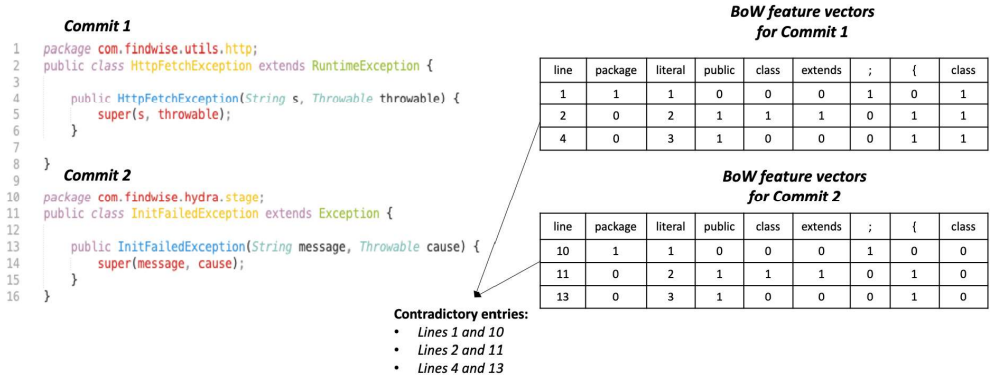


Fig. 1. Class noise in build data.

either responds by addressing the comments raised by the reviewers or arguing against the proposed changes. This feedback loop between team members remains active until the majority of the reviewers are satisfied with the discussion/modifications or until the submitted commit gets discarded.

Gerrit is an example of such tools that have recently gained popularity in OSS projects and several other Google projects (e.g., Chromium). In Gerrit, any submitted commit is only merged into the master branch if the assigned reviewers and the automatic checker have explicitly approved the changes in the submitted commits. Gerrit utilizes a voting mechanism that enables code reviewers to indicate their level of approval on merging new code changes. The voting mechanism uses a scale of intervals that span between −2 and +2. An interval of:

—+2 indicates that the change looks good and is approved.
—+1 indicates that the change looks good, but someone else must approve it.
—0 indicates no score.
—−1 indicates that it is not preferable to submit this commit.
—−2 indicates that the change is rejected.

This voting mechanism, together with the code review discussions, constitutes the evidence needed by team members to either integrate or rework the proposed change.

3.4 Class Noise Example in Code Review Data

Figure 2 exemplifies a scenario in which class noise can be introduced into a code review dataset. The figure shows two code commits written in Java—commit 1 and commit 2—submitted for peer review inspection via Gerrit. The assigned reviewers to commit 1 agree to decline the merge request—the majority of reviewers voted with a score below 0—and start a discussion thread with the author, requesting a fix to the code. On the other hand, the assigned reviewers to commit 2 agree to accept the commit—the majority of reviewers voted with a score above 0—and, hence, merge it into the master branch.

If we use the sentiment of reviewers to label each line of code in the two commits, then every line of code in commit 1 will be labeled with “disapproved” and every line of code in commit 2 will be labeled as “approved.” Note that in this example, we use a class value of “0” to annotate a line of code that is disapproved and a class value of “1” to annotate a line of code that is approved. Accordingly, the following pairs of lines from commits 1 and 2 can be classified as contradictory, since they are duplicates and hold different class labels:

—line 8 from commit 1 and line 9 from commit 2,
—line 9 from commit 1 and line 11 from commit 2, and

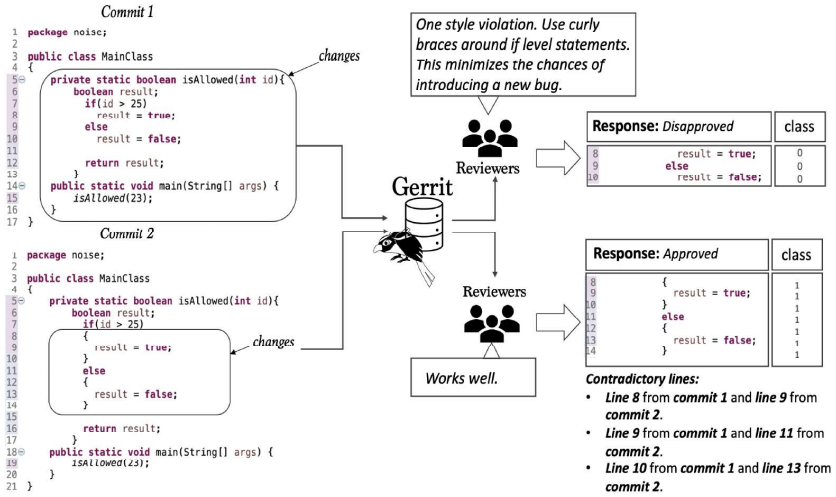


Fig. 2. Class noise in code review data.

—line 10 from commit 1 and line 13 from commit 2.

Since there are a total of 19 lines of code in the changes of commits 1 and 2, then the total number of entries in the training data is 19. The formula for calculating the noise ratio for this example is thus:

$$\text{Noise ratio} = \frac{6}{19} = 31.5\%.$$

3.5 Noise-handling Techniques

In our work, we examine the effectiveness of two removal-based and two corrective-based techniques for class noise-handling. We begin this section by describing the two removal-based techniques and then describe the corrective-based techniques.

3.5.1 Removal-based Noise-handling Techniques. From a wide range of existing class noise-handling techniques, we chose to examine two of the most widely used and reported in the literature. Namely the CF and MF introduced by Brodley and Friedl [9]. The two techniques employ an ensemble of ML models for classifying noisy entries in the training data using a voting mechanism: a univariate decision tree (C4.5), a KNN, and an LR model. Figure 3 illustrates the main procedure of the techniques for identifying and removing noisy entries. The techniques work in a k-fold cross-validation manner, where for k repetitions $k - 1$ folds are used for training each model in the ensemble. Each model is then used to tag each entry in the remaining hold-out fold as either noisy or clean. At the end of the k repetitions, each entry in the entire dataset is tagged with a label that denotes whether the entry is noisy or not. Finally, a decision about which entries should be treated as noisy—and thus removed—or not is made using a voting mechanism and accordingly. It is worth noting that CF is considered more aggressive than MF, since it removes a higher proportion of entries from the data [43].

Based on the majority voting mechanism described in [9], an entry that gets tagged as noisy by more than 50% of the models must be treated as such and thus removed from the data. On the other hand, the CF voting mechanism follows a more conservative approach suggesting that if an entry gets tagged by one or more models as noisy then it should be treated as such and removed from the dataset.

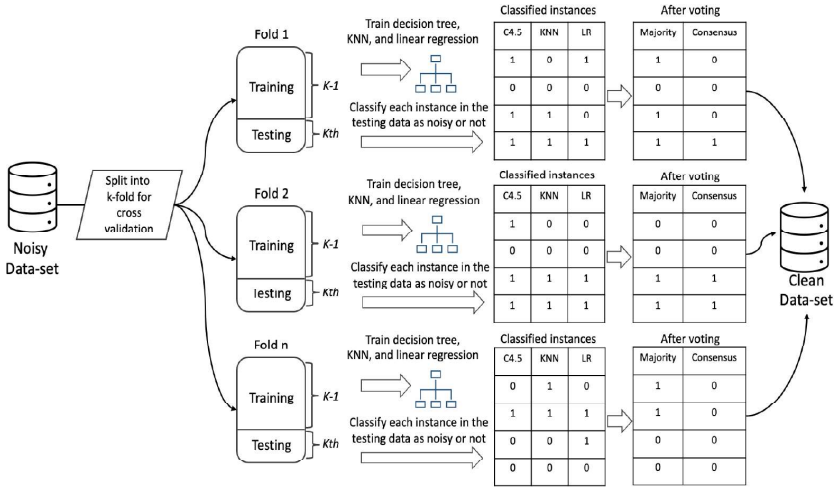


Fig. 3. Noise-handling procedure using the statistical-based techniques.

3.5.2 Corrective-based Noise-handling Techniques. We complement the analysis by examining the effectiveness of two corrective-based techniques: (1) **Domain Knowledge-based (DB)**—that we developed and published in a previous study [5], and (2) CleanLab.

Unlike statistical-based techniques (such as CF and MF), the DB was designed based on our knowledge in the domain of source code changes to correct and remove noisy entries. The procedure of the technique can be summarized by the following steps:

—Sequentially assigns a unique 8-digit hash value for each line of code in the original dataset.
—Creates an empty dictionary for storing unfiltered entries.
—Iterate through the set of hashed entries in the original dataset and save syntactically unique entries into the dictionary.
—Compare the class values between each pair of duplicate entries, both in the original and dictionary sets. If the two values are different and the value of the entry in the original set is annotated with the "positive" class, then relabel the entry in the dictionary from "negative" to "positive" and discard the entry in the original set. If both entries have the same class values, then add the entry from the original set to the dictionary.

This way of handling class noise can be seen as both corrective and removal, since it (1) corrects the class value of the same entries that first appear in the "negative" class and then the "positive class," and (2) removes one entry in each pair of contradictory entries.

In the context of code reviews and build predictions, problematic lines of code often occupy a small proportion of the overall code fragment of commits. Thus, a code fragment that was negatively perceived by a reviewer (i.e., needs to be fixed) is not likely to have issues in every line of code. Similarly, a line of code that appears as part of a passing build is not likely to trigger failure in a build job. Hence, the DB technique ensures to relabel contradictory lines of code from "disapproved" to "approved" in code reviews data and from "failing" to "passing" in build jobs data—if those lines have already been seen as part of approved reviewed fragments or passing builds.

The second examined technique in our study is a modern noise-handling technique called CleanLab. The technique was designed to improve model performance and reliability by automatically identifying and correcting mislabeled data using an unsupervised learning framework. Its design draws upon the learning theory [35] and principles of semi-supervised learning, utilizing statistical and techniques to manage label noise.

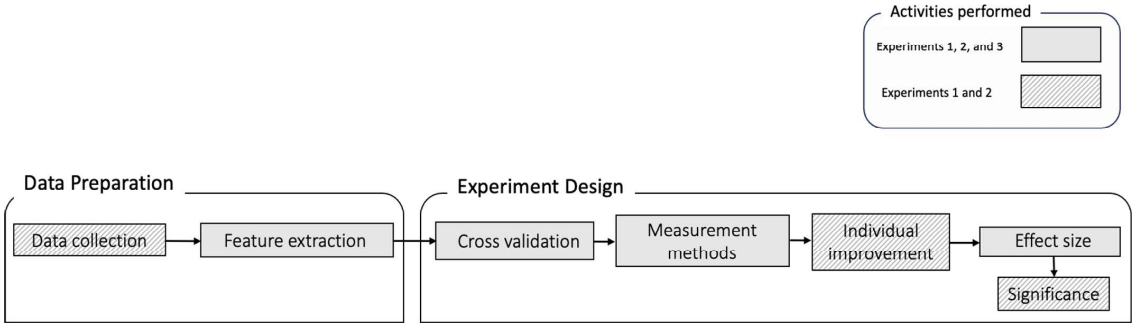


Fig. 4. Illustration of the research design activities.

The CleanLab framework provides a suite of functionalities, including mislabel detection, estimation of label noise, and correction of noisy labels.

Following the belief learning framework, CleanLab operates by first training a classification model to predict the label for each instance. It then applies confidence learning to compute a noise score, estimating the likelihood that each sample is incorrectly labeled. Based on estimated noise probabilities, CleanLab offers several strategies for correcting mislabeled data, such as removing or correcting high-noise samples or prioritize samples with low noise scores when retraining models. In this study, we utilize the correction feature offered by CleanLab for high-noise samples of lines of code.

4 Research Design

Figure 4 presents the procedure and the activities that we carried out in the three experiments. Note that the stripped blocks in Figure 4 correspond to activities that we performed in Experiments 1 and 2 only, whereas solid blocks correspond to activities performed in all of the experiments. In the following, we provide a detailed description of each activity presented in Figure 4.

4.1 Data Preparation

We begin this section by describing the independent and dependent variables of this study. Then, we describe the data collection and the feature extraction methods that we used to prepare our data.

4.1.1 Experiment Variables.

Independent Variables. noise-handling technique is the only independent variable (treatment) examined for an impact on the performance of an ML model. Four variations (treatment levels) of the independent variable are applied, namely CF, MF, DB, and CleanLab techniques. The CF, MF, and CleanLab techniques rely on statistics to detect and remove/correct noisy entries, whereas the DB technique relies on domain knowledge to relabel contradictory entries. Note that each treatment level is independent from one another and no cross-level effects are possible.

Dependent Variables. The dependent variables in this study are four state-of-the-art metrics that we use to evaluate the impact of each treatment level in experiments 1 and 2. The four metrics are Precision, Recall, F1, and MCC. We selected these metrics based on their popularity in existing state-of-the-art studies in code review and build outcome prediction. For example, several studies on build outcome and code review prediction (e.g., [1, 42, 58, 59]) have commonly used Precision, Recall, and F1-score to evaluate the prediction performance of their models. Since F1-score can sometimes lead to misleading conclusions [11], we decided to complement the analyses by using the MCC metric for evaluation. A high MCC indicates that the performance of the model is good

Table 2. Excluded Projects from the Analysis due to Class Imbalance after Applying the DB Technique

Project	Class 0	Class 1	Class ratio
azkaban	6,817	58,539	11.64%
intellij-elixir	985	242,697	0.4%
java-design-patterns	55	23,249	0.24%
jinjava	8	12,747	0.06%
jsonschema2pojo	3	15,091	0.02%
nodeclipse-1	27	23,959	0.11%
picard	378	11,017	3.43%

in predicting the binary classes. Thus, MCC considers what share of the elements in the negative class is correctly identified as negative.

In the context of build outcome prediction, Precision expresses the proportion of correctly predicted lines of code that do not trigger build failures to the total number of lines predicted as such. A high Precision means that the model performs well in predicting lines that do not trigger build failures. Recall expresses the proportion of correctly predicted lines that do not trigger build failures to the total number of lines that actually do not trigger build failures. A high Recall indicates that many of the passing builds are correctly recognized by the model.

In the context of code change request prediction, Precision expresses the proportion of correctly predicted lines of code that do not contain quality issues to the total number of lines predicted as such. Recall expresses the proportion of correctly predicted lines of code to the total number of lines in the program that actually do not contain quality issues.

The F1-score is considered as one of the most popular metrics to evaluate the classification performance of ML models [11]. It uses three elements in the confusion matrix (True positives, False positives, and False negatives) to provide a harmonic mean between Precision and Recall.

4.1.2 Collection of Build Outcomes Data (Experiment 1). The study subjects used in the first experiment were extracted from a public repository that we published in a previous work [2] The repository comprised a total of 49,040 build records that belong to 117 Java projects and their corresponding code change commits. Each record holds information about the execution outcome of a build job (passed/failed) executed against a commit. The repository also contains all added/modified code changes that were built by the CI server used in each project. In addition, the repository includes a set of feature vectors that represent the extracted code changes using the token frequency metric described in [2].

In the empirical study presented in this article, we began the analysis by using build information of all 117 Java projects. However, when applying the DB technique to the code change data we observed that the distribution of the binary classes in seven projects reached a class ratio close to [0%–100%]. Since using such imbalanced data for training would lead to creating a model that either makes pessimistic or optimistic decisions (i.e., always considering any entry as belonging to one of the classes), we decided to exclude these projects from the analysis and work on examining the impact of the class noise-handling techniques in the remaining 110 projects. Table 2 summarizes the distribution of classes in the seven excluded projects. The distribution of classes and their ratios in each analyzed project can be found in Table A7 in Appendix A. The original and curated versions of the dataset are available at Zenodo.[2]

[2]https://doi.org/10.5281/zenodo.15382708

4.1.3 Collection of Code Reviews Data (Experiment 2). The results presented in this article for answering RQ2 are based on two Java open source projects, namely "Wireshark" and "Google Sources." Our data collection process began by searching for publicly accessible projects that utilize Gerrit as their code review platform. We chose Gerrit since it provides (1) a REST API that facilitates the development of a mining tool for code review comments, and (2) a web interface that allows for tool validation.

In total, we collected code commit and code review data from 16 projects, obtained from two different sources. The first source was a public repository from which we collected 144,706 code review comments and their corresponding code changes across 15 Java-based projects [24]. This repository includes a set of JSON files that provide metadata linking review comments to specific locations within patch files, including the file name, line number, and patch version.

To make use of this information, we developed a tool that maps each review comment to the exact line of Java source code it references. This mapping process operates as follows: each review comment includes an encoded file name, which is matched to its corresponding Java source file using a pre-generated mapping. This mapping is a JSON structure generated during data extraction that associates each encoded identifier with its decoded Java file path and location within the local repository.

Once the source file is identified through this mapping, the tool uses the associated patch number to locate the specific version of the file that was under review. It then opens the file and retrieves the exact line of code referenced by the comment, based on the recorded line number. This process ensures that each comment is contextually grounded in the correct file version and line of code.

The final output is a structured dataset in which every review comment is precisely linked to:

—the exact line of Java code it refers to,
—the actual Java source file it is found in, and
—the specific patch version at the time of review.

A summary of the project names and the number of extracted lines of code from the repository can be found in Table 4.

The second source of data is a Gerrit repository that hosts the project "Google Sources."[3] To collect code review comments and their associated code changes from this repository, we developed another code mining tool. This tool exclusively retrieves either modified or added code changes. Our decision of extracting added and modified code changes only—excluding deleted code—aligns with approaches introduced in prior research studies [3—5].

The implementation procedure of the code mining tool can be summarized as follows:

(1) The tool initiates a request to the Gerrit API, querying all review, change, and file IDs within a specific repository.
(2) Subsequently, for each change and revision ID obtained in the previous step, the tool queries the Gerrit API to retrieve the IDs of all reviewed files along with their corresponding review comments.
(3) The tool sends a GET request to the Gerrit API for each reviewed file. In response, the API provides all code changes that were either added or modified within the reviewed file.
(4) Finally, the retrieved code changes, along with the corresponding code review comments acquired in step 2, are compiled and written into a CSV file. An example of how comments and the code changes are compiled into the csv file is presented in Table 3.

[3]https://gerrit-review.googlesource.com.

Table 3. An Example of Two Extracted Lines of Code and Their Corresponding Code Review Comments from the Googlesource Repository

Contents	Review comment	Filename
.data(cherryPickInput)	I wonder if it is correct (“”RESTful“”) to use URI like this for the purpose of cherry-pick:	patchsetcomplexdisclosurepanel.java
import com.google.gerrit.reviewdb.client.Change	unused import	changeutil.java

Table 4. The Study Subjects Extracted from the First Data Source

Id	Project	Number of lines of code and their review comments
1	acumos	1,305
2	android	19,977
3	asterix	22,305
4	carbonrom	28
5	cloudera	7,855
6	eclipse	16,602
7	fd.io	830
8	gerrithub	1,978
9	googlereview	22,837
10	iotivity	1,244
11	omnirom	358
12	opencord	89
13	polarsys	371
14	unicorn	25
15	wireshark	48,902

Applying this tool to the “Google Sources” repository resulted in the collection of a total of 74,003 lines of code and their associated review comments. To ensure the accuracy of our tool, we cross-verified the review comments of two file IDs in a revision against those that we manually extracted using the Gerrit web interface.

Labeling Code Review Comments. After collecting the datasets, we decided to manually label a sample of the collected code comments from each projects. The labeling guideline was as follows:

—If a code comment explicitly requested a change to the code, it was labeled as “0.”
—If a code comment accepted the change or expressed praise for it, it was labeled as “1.”
—If a code comment suggested an optional change or raised a question, it was labeled as “neutral.”

Table 5 presents examples of sentiment labels corresponding to review comments, illustrated using three code snippets from the Wireshark project.

To ensure inter-annotator agreement and to refine the annotation guidelines, we conducted a pilot annotation phase. In this step, the three authors of this article independently annotated a sample of 200 code review comments. The sample was randomly selected and included a mix of comments that requested changes and those that approved the code for integration. The comparison between annotations revealed a 95% agreement for comments requesting changes and an 80% agreement for

Table 5. Examples of Annotated Lines of Code Based on Review Sentiment

Contents	Review comment	Sentiment	File location
if ((octets_to_next_header == 0) && (version > = 0x0200) && (submessageId != SUBMESSAGE_PAD) && (submessageId != SUBMESSAGE_INFO_TS))	AS this condition is so long could you use for clarity please? _ Regards _ Anders	neutral	epan/dissectors/packet-rtps.c
DIS_FIELD_UDL(tree, offset)_	It seems like an offset++ is />lacking here	0	epan/dissectors/packet-gsm_sms.c
for line in contents:	Nice!	1	tools/check_dissector_urls.py

Table 6. The Distribution of the Annotated Comments with Respect to the "0," "1," and "Neutral" Classes

Id	Project	Class 0	Class 1	Class neutral
1	unicorn	16	1	8
2	wireshark	267	110	67
3	googlesources	166	66	62
4	asterix	341	5	142
5	googlereview	276	0	96
6	acumos	405	8	57
7	iotivity	11	2	131
8	fd.io	284	9	95
9	android	329	12	152
10	carbonrom	13	2	14
11	opencord	53	0	33
12	omnirom	131	0	41
13	gerrithub	291	9	94
14	eclipse	504	84	188
15	polarsys	167	0	28
16	cloudera	377	5	113

comments expressing approval. Based on these levels of agreement, we considered the annotation guidelines sufficiently robust and proceeded with annotating a larger sample of the dataset.

Following the pilot, the three authors collectively annotated a total of 6,255 code comments, with each author responsible for approximately one-third of the annotations. After completing the annotation process, we analyzed the class distribution of labeled comments and observed that, on average, 95% of comments fell into the "0" and "neutral" classes, as summarized in Table 6.

During the annotation process, particular attention was given to cases where a code change received both positive and negative review comments. In such cases, we applied a majority-label approach based on the overall sentiment and intent of the discussion. If most comments indicated that changes were required, the code change was labeled as requiring revision. Conversely, if the majority of feedback was approving and any concerns raised were minor or resolved during the discussion, the change was labeled as approved. For ambiguous cases where the sentiment could not be resolved based on the majority or context, the annotators discussed the example and reached a consensus. These edge cases also contributed to refining our annotation guidelines to ensure consistency and clarity in labeling.

Given the imbalanced distribution of labeled comment classes across projects, we decided to focus on a subset of projects with more balanced review discussions. Among the 16 collected projects, only two—Wireshark and Google Sources—contained a relatively even distribution of

Table 7. The Distribution of Annotated Code Comments

Project	Class 0	Class 1
Wireshark	1,665	897
Google Sources	1,552	894

comments requesting and approving changes. As such, we utilized the data extracted from these two projects to address RQ2. A total of 2,562 code comments were annotated for the Wireshark project, and 2,446 comments for the Google Sources project. The distribution of the annotated comments is presented in Table 7.

4.1.4 Dataset for Examining the Impact of BoW and CodeBERT (Experiment 3). To address RQ3, we analyzed a sample of 11 projects from the dataset collected to answer RQ1 and RQ2. This subset comprised nine projects from the build dataset that we utilized in Experiment 1 and the two projects in Experiment 2. The selection of the sample was guided by the MCC scores initially achieved by the model after applying each class noise-handling technique to the training data for build outcomes. In particular, among the selected nine projects from the build dataset, three projects demonstrated a considerably high MCC after being exposed to the three class noise-handling techniques. Another three projects demonstrated a notable decrease in MCC when applying DB to the training data, yet exhibited noteworthy improvement when subjected to MF and CF. The last set of projects demonstrated a relatively low MCC after applying the three class noise-handling techniques to the training data. The project names of the selected sample from the build dataset are highlighted in bold in Table A2.

4.1.5 Dataset for Examining the Impact of Hyperparameter Tuning (Experiment 4). We implemented the fourth experiment using a sample of 14 datasets from the build data used to address RQ1. The selection of these datasets was guided by the MCC scores of the model before applying any noise-handling techniques, as reported in our previous study [2]. Specifically, the criterion for selecting the sample datasets was to choose the datasets where the model's MCC scored lower than zero (MCC < 0). This criterion was motivated by the fact that a negative MCC score indicates that the model's predictions are worse than random guessing. By selecting datasets with negative MCC scores, we aimed at investigating the impact of hyperparameter tuning in scenarios where the baseline model produced false guessing of build execution outcomes.

4.1.6 Feature Extraction. In this study, we employ two textual analysis techniques that extract features from a corpus of code changes. The first technique is based on the BoW model and is used in our analysis of the three experiments. The second technique is CodeBERT, which we use to examine its impact on the performance of noise-handling techniques in the third experiment.

BoW. Each feature extracted with this technique corresponds to a code token that appears in the extracted code. In this study, we utilize a tool proposed by Ochodek et al. [36] to perform the features extraction using the BoW model. The textual analysis tool follows the following procedure to extract features:

— creates a vocabulary for all lines of code (using the BoW technique, with a cutoff parameter of how many words should be included[4]),

[4]BoW is essentially a sequence of tokens, which are descendingly ordered according to frequency. This cutoff parameter 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.

— creates a token for words that fall outside of the frequency defined by the cutoff parameter of the BoW,
— finds a set of predefined keywords in each line, and
— checks each word in the line to decide if it should be tokenized or if it is a predefined feature.

The output of this step is a large array of numbers, each representing the token frequency of a specific feature in the BoW space of vectors. In this study, we chose to use a bigram model to represent the feature vectors, as it was previously shown to produce good learning performance in a similar context (e.g., [5]). The raw experimental data and BoW feature vectors are available (see footnote 2) .

CodeBERT. A context-dependent technique that generates contextual embeddings where the same token in the input source code can have different representations based on the different surrounding context in the corpus. CodeBERT is pre-trained on a large dataset of GitHub open source code [15]. During training, the model learns to predict missing parts of code snippets by understanding the context and relationships between different code tokens. The architecture of CodeBERT follows that used in BERT [13] and RoBERTa [41], where a multilayer bidirectional transformer is adopted. In Experiment 3, we used CodeBERT for examining its impact on noise-handling for several purposes, compared to BoW. First, CodeBERT is pre-trained on a large corpus of source code, enabling it to capture domain-specific knowledge related to programming languages, syntax, and semantics. Second, CodeBERT-based solutions have reported promising results in terms of their effectiveness compared to other General-purpose-based Transformer solutions, which can be computationally more demanding [61].

4.2 Design of the Experiments

The overarching goal of this study is to examine the effect of class noise-handling techniques on the performance of a model for predicting build outcome and negative code review comments within CI. We address this goal by examining RQ1 and RQ2, which focus on the impact of noise-handling techniques. Therefore, we use the same activities in the design and execution of the first two experiments to address RQ1 and RQ2. In particular, the following five activities are carried out:

(1) cross validation,
(2) measurement method,
(3) analysis of individual improvement,
(4) analysis of effect size, and
(5) analysis of significance testing.

Additionally, RQ3 investigates the impact of different feature extraction techniques on the efficacy of class noise-handling, while RQ4 examines the impact of hyperparameter tuning on the model's predictive performance for build outcome prediction. In Experiments 3 and 4, we perform activities 1, 2, and 4 to answer RQ3 and RQ4.

4.2.1 Cross Validation. We applied a 10-fold stratified cross validation partitioning scheme on each dataset in both experiments, i.e., 10 random partitions of each project dataset with a combination of nine of them (90%) as training set and the remaining one as a test set (10%). A total of 1,100 training and testing trials were carried out in Experiment 1, and 160 training and testing trials were carried out in Experiment 2. The distribution of the classes in each training fold is then evaluated to decide whether subsequent balancing of the classes for each training fold is required. If the distribution of one class exceeds 60%, then the minority class is over-sampled until all data

Table 8. The Hyperparameter Space Examined for an Impact on the Predictive Performance of RF Model for Build Outcomes

Hyperparameter	Range
n_estimators	100, 200, 300
max_depth	None, 10, 20, 30
min_samples_split	2, 5, 10
min_samples_leaf	1, 2, 4
bootstrap	True, False

points in the minority class even out with the number of entries in the majority class. We used the generated testing folds from the original data to evaluate the performance of the ML model before and after applying each treatment level on the training folds, respectively.

4.2.2 Measurement Method. To examine the impact of the four class noise-handling techniques, we conducted a preliminary evaluation to compare and select the most suitable model for the prediction of a sample of 12 datasets and 3 classifiers: RF, **Extreme Gradient Boosting (XGBoost)**, and a two-dense **Neural Network (NN)**. The goal of this evaluation was to identify a suitable classifier with the highest predictive performance for measuring the impact of the class noise-handling techniques.

We selected these classifiers based on their unique characteristics and effectiveness in tolerating noise in data. Particularly, we chose RF due for its robustness in predicting similar SE tasks, such as test case selection [5]. Similarly, XGBoost is widely recognized for its high predictive performance and scalability in different applications [7]. Additionally, NNs are renowned for their ability to learn intricate representations within the data [48].

During our preliminary evaluation of effective models, we kept the RF and XGBoost models at their default parameter values as provided by the scikit-learn library (version 0.20.4) [38]. The NN model was a sequential model, consisting of two dense layers, and was implemented using the Keras library [12]. Each of the three evaluated models was trained on both the experimental data (exposed to the treatment) and the control data (before applying the treatment). The evaluation results are presented in Table A1 in Appendix A. The results revealed that RF outperformed the other two models when trained on the CF and MF experimental data, as well as when trained on the control data. Consequently, we decided to select RF as the primary method for measuring the impact of the four class noise-handling techniques on the predictive performance of a model for build outcome and code change request predictions.

To further gain insights into the impact of class noise-handling in the context of CI, we extended our analysis by examining the impact of hyperparameter tuning of the RF model on the predictive performance of negative code review comments. To that end, we conducted an experiment wherein we utilized a Grid Search algorithm to optimize the hyperparameters of the RF model. The grid search algorithm was chosen due to its wide acceptance and use in ML studies [55]. The RF hyperparameter space that we explore for impact is shown in Table 8. For each hyperparameter, the table shows the examined range of values.

4.2.3 Individual Improvement. To understand the impact of each class noise-handling technique on the predictive performance of a model for build outcome and negative review comment predictions, we examine improvement trends in the four dependent variables before and after applying the treatment on each study subject. Another important measurement that we make is the ratio of

Table 9. The Hypotheses for the Effects of Noise-handling Techniques on Build Outcome and Code Change Request Predictions

Precision	**Recall**	**F1**	**MCC**
H_{0p_m}: The mean Precision is the same for a model trained on MF-curated and non-curated data. $\mu_{0p\_m} = \mu_{1p\_m}$	H_{0r_m}: The mean Recall is the same for a model trained on MF-curated and non-curated data. $\mu_{0r\_m} = \mu_{1r\_m}$	H_{0f_m}: The mean F1 is the same for a model trained on MF-curated and non-curated data. $\mu_{0f\_m} = \mu_{1f\_m}$	H_{0m_m}: The mean MCC is the same for a model trained on MF-curated and non-curated data. $\mu_{0m\_m} = \mu_{1m\_m}$
H_{0p_c}: The mean Precision is the same for a model trained on CF-curated and non-curated data. $\mu_{0p\_c} = \mu_{1p\_c}$	H_{0r_c}: The mean Recall is the same for a model trained on CF-curated and non-curated data. $\mu_{0r\_c} = \mu_{1r\_c}$	H_{0f_c}: The mean F1 is the same for a model trained on CF-curated and non-curated data. $\mu_{0f\_c} = \mu_{1f\_c}$	H_{0m_c}: The mean MCC is the same for a model trained on CF-curated and non-curated data. $\mu_{0m\_c} = \mu_{1m\_c}$
H_{0p_d}: The mean Precision is the same for a model trained on DB-curated and non-curated data. $\mu_{0p\_d} = \mu_{1p\_d}$	H_{0r_c}: The mean Recall is the same for a model trained on DB-curated and non-curated data. $\mu_{0r\_d} = \mu_{1r\_d}$	H_{0f_d}: The mean F1 is the same for a model trained on DB-curated and non-curated data. $\mu_{0f\_d} = \mu_{1f\_d}$	H_{0m_d}: The mean MCC is the same for a model trained on DB-curated and non-curated data. $\mu_{0m\_d} = \mu_{1m\_d}$
H_{0p_d}: The mean Precision is the same for a model trained on cleanlab-curated and non-curated data. $\mu_{0p\_cl} = \mu_{1p\_cl}$	H_{0r_c}: The mean Recall is the same for a model trained on cleanlab-curated and non-curated data. $\mu_{0r\_cl} = \mu_{1r\_cl}$	H_{0f_d}: The mean F1 is the same for a model trained on cleanlab-curated and non-curated data. $\mu_{0f\_cl} = \mu_{1f\_cl}$	H_{0m_d}: The mean MCC is the same for a model trained on cleanlab-curated and non-curated data. $\mu_{0m\_cl} = \mu_{1m\_cl}$

class noise, as defined in Section 3.1, before and after applying the treatment to the training data. The individual impact of each treatment level on the four dependent variables and class noise ratio is visually analyzed using scatter plots.

4.2.4 Effect Size. To summarize the effect of the four noise-handling techniques on the predictive performance of the RF model in each experiment, we calculate the descriptive statistics of the four dependent variables before and after applying the treatment to the training data. The effect of each technique is visualized by plotting the mean scores and distributions of the four dependent variables.

4.2.5 Significance Testing. We hypothesize that using any of the four class noise-handling techniques, described in Section 3.1, would improve the performance of a model for predicting negative code review comments and build outcomes, compared to when leaving noisy entries in the training data. Accordingly, 12 hypotheses are formally defined and tested for statistical significance in each experiment. Table 9 lists and defines the hypotheses.

All of the hypotheses listed in Table 9 are two-tailed, since we are mainly interested in understanding whether the value of each dependent variable would be impacted by any of the four class noise-handling techniques. Each hypothesis is defined in terms of one variation of the independent and dependent variables. For example, the first hypothesis under column "Precision" suggests that applying MF (a variation of the independent variable) on the training data will result in a

significantly different Precision (a dependent variable) score compared to when leaving noisy entries in the training data.

Normality Test. To determine whether to use parametric or non-parametric statistical tests, we checked for the normality assumption of the four dependent variables using the Shapiro-Wilk test available in the scikit-learn library [38]. The results showed that the distribution of the four dependent variables was not normally distributed. Based on the normality test results, we decided to run the Kruskal-Wallis (a non-parametric test) for comparing the Precision, Recall, F1-score, and MCC values between the different treatment levels. The Mann-Whitney U test was then run to perform a pairwise comparison between the dependent variables under each treatment level and the same measures that we recorded after training on the original dataset.

5 Results

This section reports the results of the two experiments to answer the two RQs.

5.1 The Impact of Class Noise-handling on Predicting the Outcome of Builds in CI (RQ1)

To address our RQ of *What is the impact of applying class noise-handling on predicting the outcome of builds in CI*?, we structure our results into three analyses:

—Individual improvements per noise-handling technique and dataset, Section 5.1.1.
—Effect size of the improvements per technique, Section 5.1.2.
—Significance analysis of the effects per technique, Section 5.1.3.

5.1.1 Individual Improvements. Figure 5 illustrates the impact of each technique on the Precision measure across the examined study subjects. In general, we can observe that the two removal techniques (MF and CF) consistently improve the Precision measure whilst reducing the ratio of class noise in the majority of the study subjects. This observation is supported by the diagonal upward trend observed in most of the lines depicted in Figure 5(a) and (b). Specifically, we can observe that:

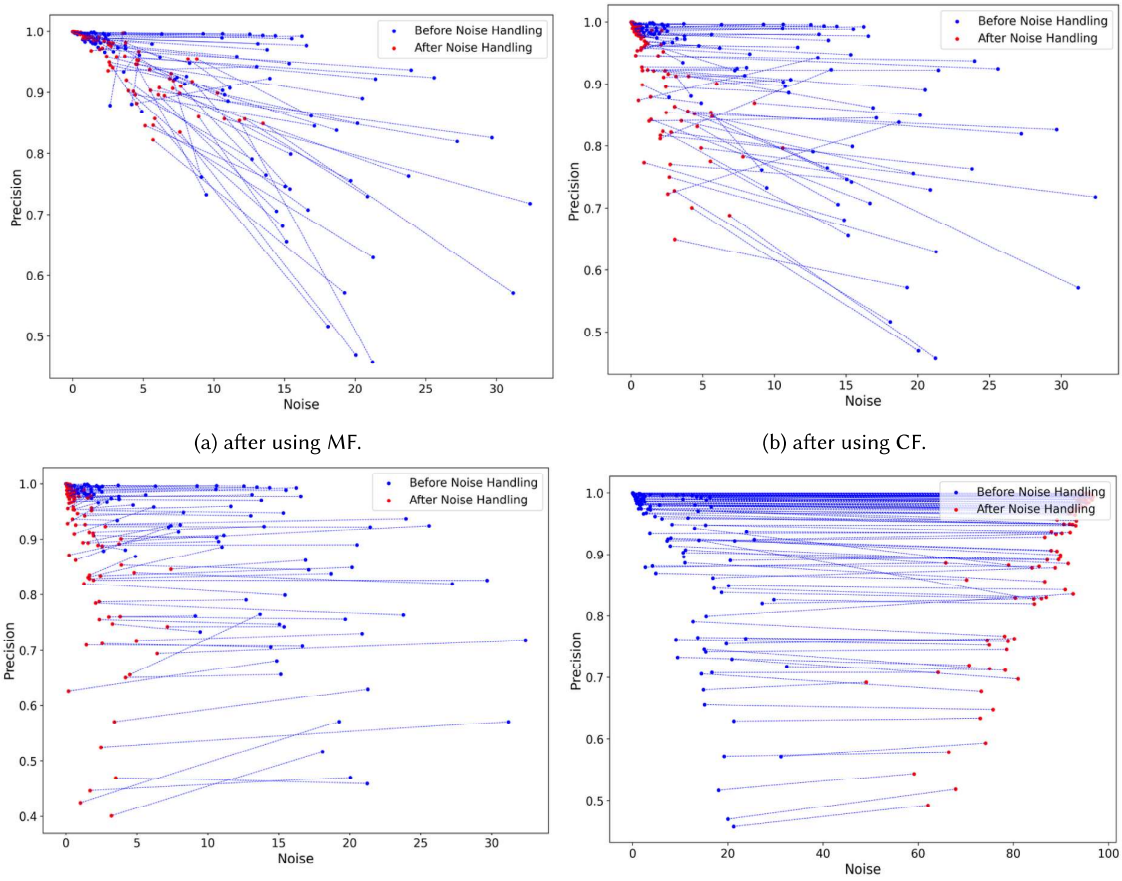
—MF resulted in a Precision improvement for 100/110 study subjects.
—CF resulted in a Precision improvement for 74/110 study subjects.
—DB resulted in a Precision improvement for 34/110 study subjects.
—CleanLab resulted in a Precision improvement for 24/110 study subjects.

These observations imply that using the MF or CF techniques would consistently lead to an improvement in Precision.

Another notable observation is the impact of applying class noise-handling techniques on the class noise ratio—the number of contradictory entries to the overall size of the study subjects. In this regard, MF, CF, and DB consistently reduce class noise across all study subjects. In contrast, CleanLab did not lead to a reduction in the overall class noise ratio in any of the examined study subjects. Specifically, we found that:

—MF reduced the ratio of class noise in 89/110 study subjects.
—CF reduced the ratio of class noise in 103/110.
—DB reduced the ratio of class noise in all study subjects.

Figure 6 illustrates the impact of each technique on the Recall measure across the examined study subjects. In general, we can observe that MF and CF are consistent in the effect and have



(a) after using MF.

(b) after using CF.

(c) after using DB.

(d) after using CleanLab.

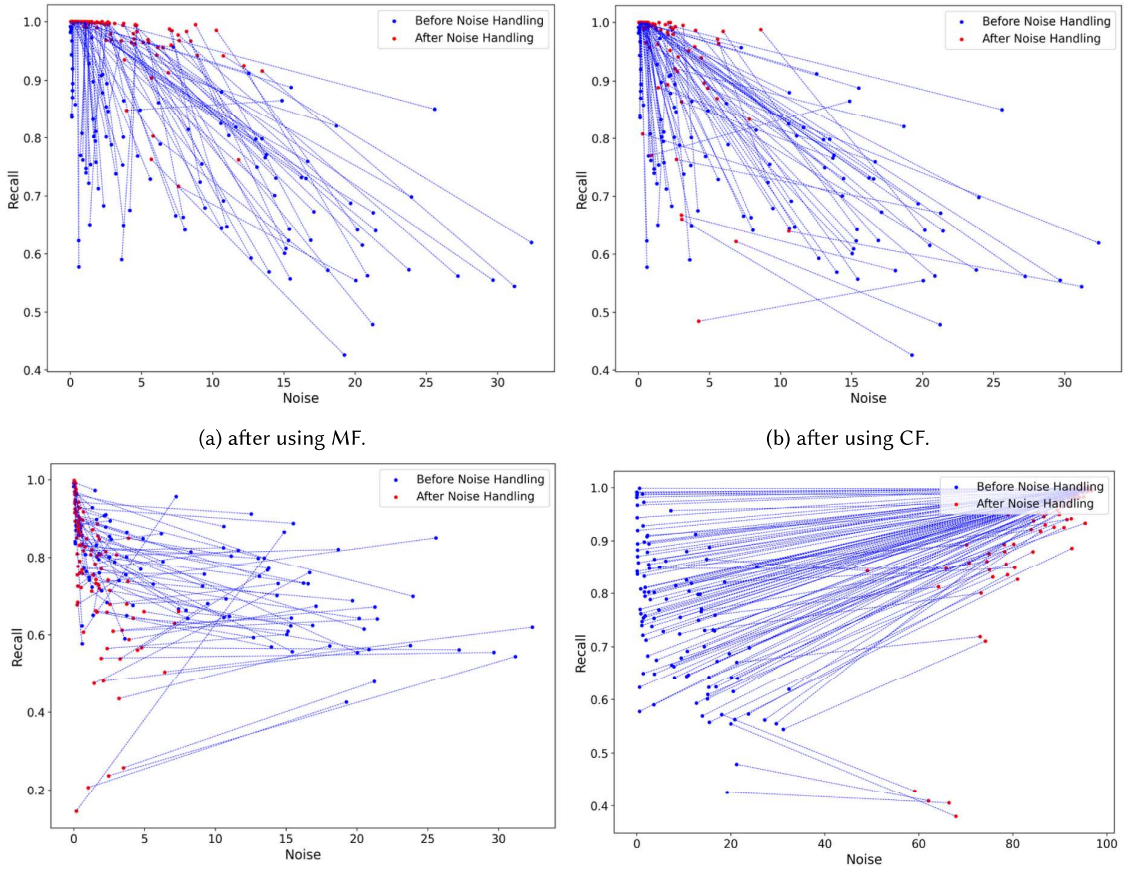
Fig. 5. Precision scores after applying the treatment. The x-axis corresponds to the ratio of class noise, whereas the y-axis corresponds to the mean Precision scores. Blue marks represent the mean Precision scores before applying the treatment and the ratio of class noise. Red marks represent the mean performance score after applying the treatment and the new ratio of class noise.

a positive impact on Recall for the prediction of the majority of study subjects—most lines are diagonal in an upward direction. On the other hand, DB is less consistent in its effect on Recall. Specifically, we found that:

—MF resulted in a Recall improvement for 109/110 study subjects.
—CF resulted in a Recall improvement for 108/110 study subjects.
—DB resulted in a Recall improvement for 76/110 study subjects.
—CleanLab resulted in a Recall improvement for 102/110 study subjects.

The above observations imply that applying any of the two removal-based techniques, MF and CF, lead to a more consistent improvement in Recall compared to DB and CleanLab.

Figure 7 illustrates the impact of each technique on the F1 measure across the examined study subjects. Consistent with the findings observed for Precision and Recall, it can be observed that both MF and CF lead to a consistent improvement in F1. In contrast, applying DB does not consistently lead to an improvement in F1. Specifically, we found that:



(a) after using MF.

(b) after using CF.

(c) after using DB.

(d) after using CleanLab.

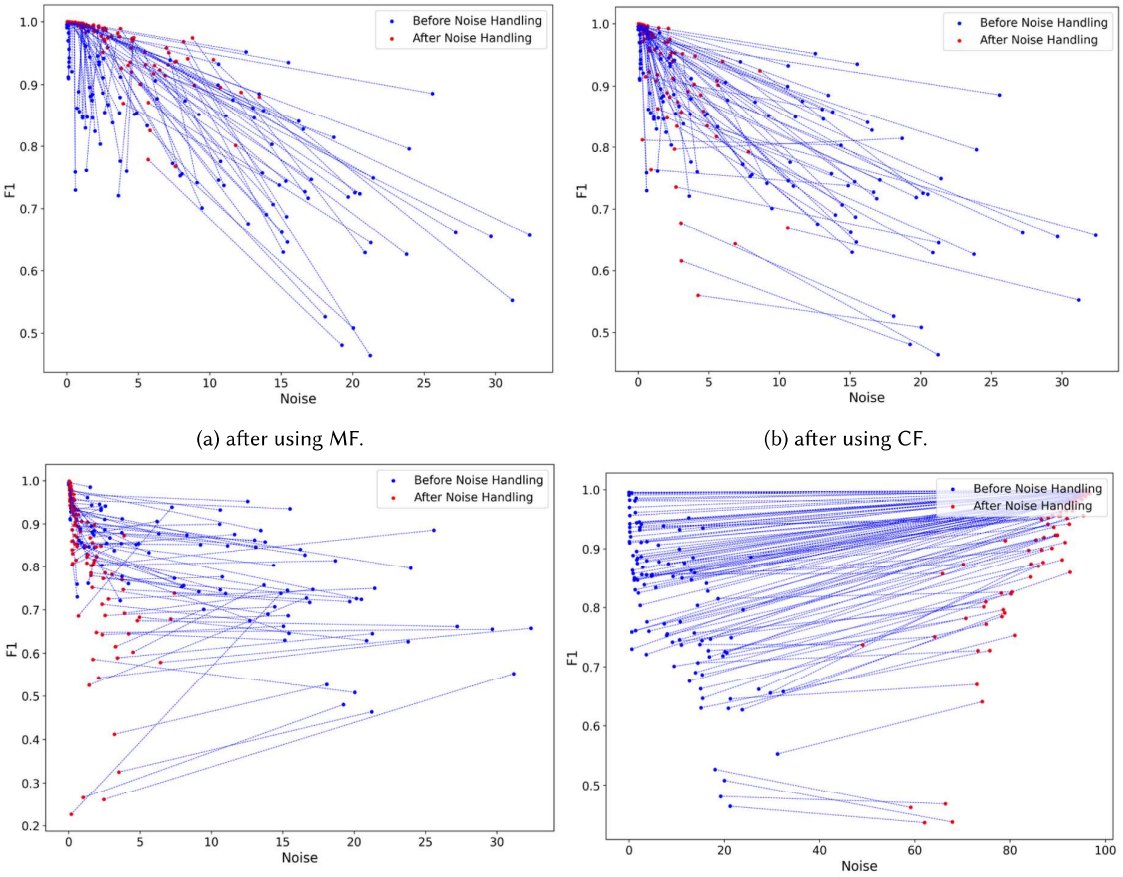
Fig. 6. Recall scores after applying the treatment. The x-axis corresponds to the ratio of class noise, whereas the y-axis corresponds to the mean Precision scores. Blue marks represent the mean Precision scores before applying the treatment and the ratio of class noise. Red marks represent the mean performance score after applying the treatment and the new ratio of class noise.

—MF resulted in an F1 improvement for 110/110 study subjects.
—CF resulted in an F1 improvement for 109/110 study subjects.
—DB resulted in an F1 improvement for 76/110 study subjects.
—CleanLab resulted in an F1 improvement for 103/110 study subjects.

The above observations suggest that the two removal-based techniques, MF and CF, lead to more consistent improvements in F1 compared to both DB and CleanLab.

Figure 8 presents the impact of each technique on the MCC measure for each study subject. From the figure, we can observe that all techniques are consistent in the effect on MCC, but in opposite directions. In general, we can observe that both MF and CF exhibit a consistent positive impact on MCC, while DB consistently leads to a deterioration of MCC.

—MF resulted in an MCC improvement for 110/110 study subjects.
—CF resulted in an MCC improvement for 108/110 study subjects.
—DB resulted in an MCC improvement for 33/110 study subjects only.



(a) after using MF.

(b) after using CF.

(c) after using DB.

(d) after using CleanLab.

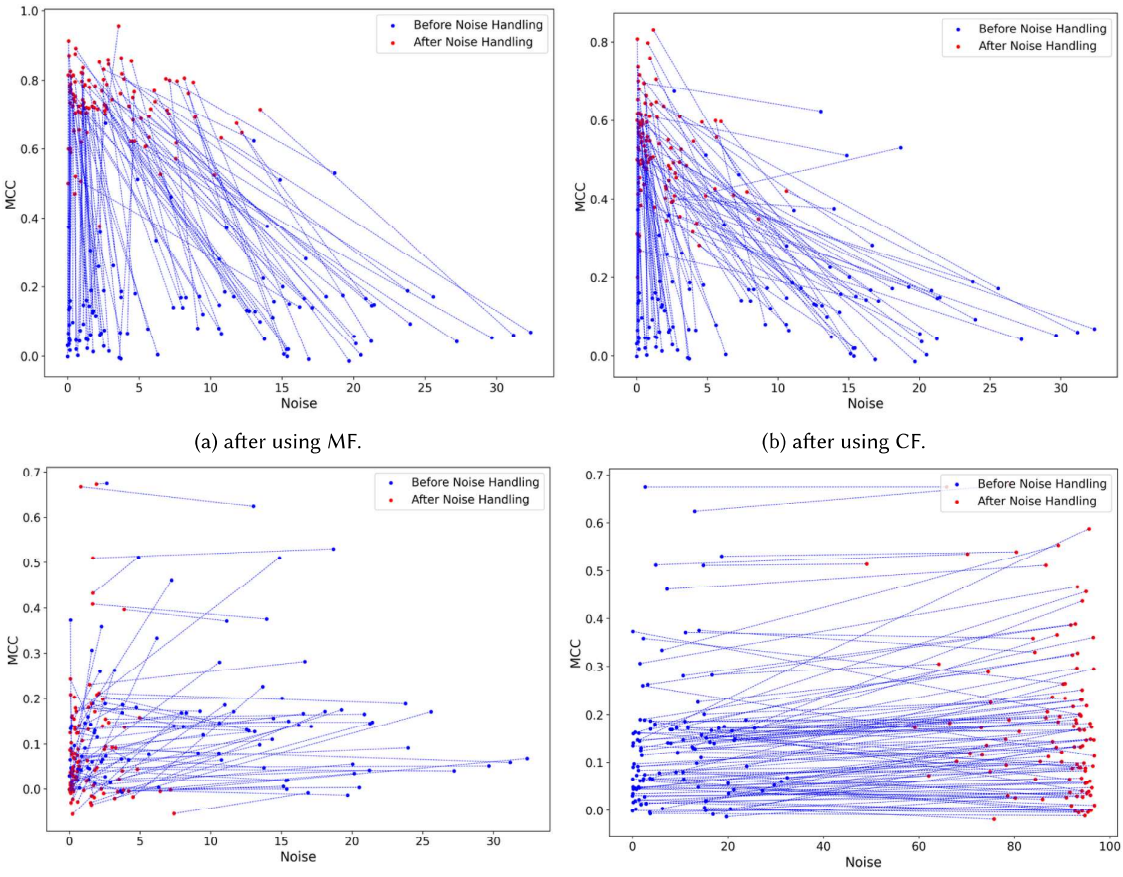
Fig. 7. F1 scores after applying the treatment. The x-axis corresponds to the ratio of class noise, whereas the y-axis corresponds to the mean Precision scores. Blue marks represent the mean Precision scores before applying the treatment and the ratio of class noise. Red marks represent the mean performance score after applying the treatment and the new ratio of class noise.

—CleanLab resulted in an MCC improvement for 81/110 study subjects only.

These observations imply that the removal-based techniques, MF and CF, consistently yield greater improvements in MCC compared to both DB and CleanLab. While removal methods generally outperform correction-based approaches in improving the predictive performance of build outcomes, it is noteworthy that CleanLab demonstrated considerably higher predictive performance gains in Recall, F1, and MCC than DB, despite showing minimal reduction in the overall class noise ratio across most study subjects.

5.1.2 Descriptive Statistics. Figure 9 is a bar plot that visualizes the mean percentages of Precision, Recall, F1, and MCC scores before and after applying the four treatment levels respectively. The x-axis represents the treatment levels and the y-axis corresponds to the values of the four dependent variables.

Applying the four class noise-handling techniques on the control group data had the following impact on Precision, on average:



(a) after using MF.

(b) after using CF.

(c) after using DB.

(d) after using CleanLab.

Fig. 8. MCC scores after applying the treatment. The *x*-axis corresponds to the ratio of class noise, whereas the *y*-axis corresponds to the mean Precision scores. Blue marks represent the mean Precision scores before applying the treatment and the ratio of class noise. Red marks represent the mean performance score after applying the treatment and the new ratio of class noise.

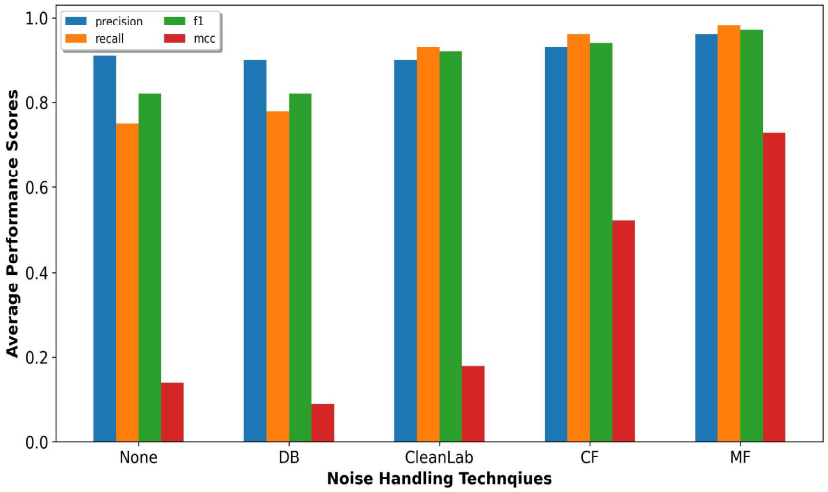


Fig. 9. Mean Precision, Recall, F1, and MCC after using each noise-handling technique on build outcomes data.

—MF resulted in improving Precision from 91% to 96%.
—CF resulted in improving Precision from 91% to 93%.
—DB resulted in a slight decrease in Precision from 91% to 89%.
—CleanLab resulted in a slight decrease in Precision from 91% to 90%.

Applying any of the four class noise-handling techniques to the control group data resulted in improved Recall values. Specifically, on average:

—MF resulted in improving Recall from 76% to 98%.
—CF resulted in improving Recall from 76% to 96%.
—DB resulted in improving Recall from 76% to 78%.
—CleanLab resulted in improving Recall from 76% to 93%.

Applying any of the four class noise-handling techniques to the control group data resulted in improved F1 values. Specifically, on average:

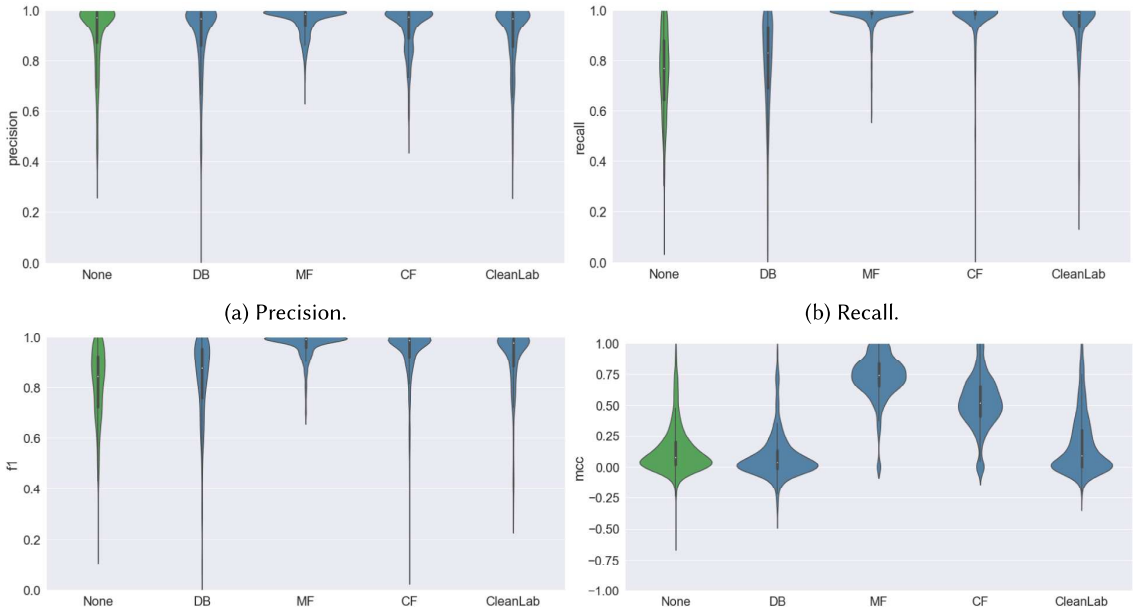
—MF resulted in improving F1 from 82% to 97%.
—CF resulted in improving F1 from 82% to 94%.
—DB resulted in no improvement in F1.
—CleanLab resulted in improving F1 from 82% to 92%.

Applying the four class noise-handling techniques on the control group data had the following impact on MCC, on average:

—MF resulted in improving MCC from 0.13 to 0.58.
—CF resulted in a improving MCC from 0.13 to 0.52.
—DB resulted in a decrease in MCC from 0.13 to 0.08.
—CleanLab resulted in improving MCC from 0.13 to 0.18.

To gain a better understanding of the impact of each class noise-handling technique, we plotted the distribution of each dependent variable for all the study subjects. The violin-plot graphs, namely Figure 10(a)–(d), illustrate the distribution of the dependent variables for the four treatment levels (MF, CF, DB, and CleanLab) as well as the control group data, labeled as "none." We identified four observations in these graphs:

—Applying MF, CF, and CleanLab led to a more compact distribution of Precision values, with most scores concentrated around the median, indicating reduced variability and improved reliability in model precision.
—After applying MF and CleanLab, the distributions of Recall, F1, and MCC scores showed decreased dispersion, suggesting improved quality of the training data and more consistent predictive performance.
—In contrast, applying DB and CF caused the F1 score distribution to become more spread out, resulting in less consistent predictions across study subjects.
—Applying any treatment level reduced the dispersion of MCC values, indicating higher stability and uniformity in model performance.
—Applying MF and CF resulted in MCC values above 0.5 in most cases, indicating a clear improvement in prediction performance.
—The post-treatment with CleanLab showed that the MCC distribution was more concentrated in the positive range (above 0), indicating increased consistency across different study subjects.
—The reduced variance in MCC values suggests that the model's behavior became more stable and less sensitive to noise after applying MF, CF, and CleanLab.



(a) Precision. (b) Recall.

(c) F1. (d) MCC.

Fig. 10. Distribution of dependent variables for the four noise-handling techniques.

These observations were further supported by examining the descriptive statistics of the dependent variables summarized in Tables A2, A3, A4, A5, and A6.

In general, our findings collectively confirm that applying the removal-based treatments (MF and CF) lead to increased predictive performance. Notably, applying MF and CleanLab lead to a more uniform score distribution across the examined performance metrics, compared to the uniformity achieved when applying DB and CF. In addition, the results suggest that applying DB improves the prediction performance for builds that will pass (improved Recall and F1), but it tends to negatively affect the prediction performance for builds that will fail (lower MCC).

5.1.3 Hypotheses Testing. To evaluate the hypotheses, we begin by checking the assumption of normality for the distribution of the four dependent variables. The Shapiro-Wilk test was carried out for testing the assumption of normality. As can be seen from Table 10, the null hypotheses of normality for the four dependent variables can be rejected (p-value < 0.05). Since we have issues with normality in the four dependent variables, we decided to run a non-parametric test for comparing the difference between each dependent variable under the four treatment levels and the control group.

Table 11 summarizes the statistical comparison results between the four dependent variables for the four treatment levels and the control group using the Kruskal-Wallis analysis test. Each column provides the test statistics and the associated p-value for one dependent variable separately. The results in Table 11 reveal that there is a statistically significant difference between the four dependent variables ($p < 0.05$).

Table 12 presents the results of conducting the Mann-Whitney test to evaluate the 16 hypotheses in Experiment 1. The pairwise comparisons between the four dependent variables indicate a significant difference between the control group and the experimental subjects exposed to MF (Precision: statistics = 4,650, Recall: statistics = 420, F1: statistics = 974, MCC: statistics = 444, all with $p = 0.05$) and CleanLab (Precision: statistics = 635,716 Recall: statistics = 172,802 F1: statistics = 285,375 MCC: statistics = 4,969, all with $p < 0.05$). These findings provide statistical evidence to reject the null

Table 10. The Shapiro-Wilk Analysis Results for Each Dependent Variable after Applying the Noise-handling Techniques to the Build Data

Dependent variables	Control group	DB	CF	MF	CleanLab
Precision	Stats = 0.74, p < 0.05	Stats = 0.73, p < 0.05	Stats = 0.8, p < 0.05	Stats = 0.79, p < 0.05	Stats = 0.75, p < 0.05
Recall	Stats = 0.98, p < 0.05	Stats = 0.89, p < 0.05	Stats = 5, p < 0.05	Stats = 0.5, p < 0.05	Stats = 0.55, p < 0.05
F1	Stats = 0.95, p < 0.05	Stats = 0.83, p < 0.05	Stats = 0.68, p < 0.05	Stats = 0.69, p < 0.05	Stats = 0.67, p < 0.05
MCC	Stats = 0.82, p < 0.05	Stats = 0.75, p < 0.05	Stats = 1.0, p < 0.95	Stats = 0.96, p < 0.05	Stats = 0.85, p < 0.05

Table 11. Statistical Comparison Results between the Dependent Variables after Applying the Treatment Levels on the Build Data

	Precision	Recall	F1	MCC
Kruskal-Wallis H	173.72	1,602.73	925.48	2,563.97
Sig.	p < 0.05	p < 0.05	p < 0.05	p < 0.05

Table 12. Pairwise Comparison between the Dependent Variables for Each Treatment Level and the Control Group Using the Mann-Whitny U Test

	DB	CF	MF	CleanLab
Precision	Stats = 6,147, p < 0.49	Stats = 5,619, p < 0.64	Stats = 4,650, p < 0.05	Stats = 635,716, p < 0.05
Recall	Stats = 4,712, p < 0.05	Stats = 829, p < 0.05	Stats = 420, p < 0.05	Stats = 172,802, p < 0.05
F1	Stats = 5,020, p < 0.07	Stats = 1,831, p < 0.05	Stats = 974, p < 0.05	Stats = 285,375, p < 0.05
MCC	Stats = 8,015, p < 0.05	Stats = 448, p < 0.05	Stats = 58, p < 0.05	Stats = 4,969, p < 0.05

hypotheses H_{0p_m}, H_{0r_m}, H_{0f_m}, *and* H_{0m_m}, H_{0p_cl}, H_{0r_cl}, H_{0f_cl}, *and* H_{0m_cl} suggesting that MF and CleanLab have a significant impact on the predictive performance of the model for Build outcome prediction.

Similarly, the results suggest that there is a statistically significant difference between the Recall, F1, and MCC scores achieved when training a model on the control group subjects and the experimental subjects exposed to CF (Recall: statistics = 829 with, F1: statistics = 1,831, MCC: statistics = 58 with $p < 0.05$). We also found a statistically significant difference between the Recall and MCC achieved when training on the control group subjects and the experimental subjects exposed to DB (Recall: statistics = 4,712 with, MCC: statistics = 7,698 with $p < 0.05$). These results suggest that there is statistical evidence to support the rejection of the null hypotheses H_{0r_d}, H_{0r_c}, H_{0f_c}, H_{0m_d}, *and* H_{0m_c}. However, no statistical evidence was found to support the rejection of the hypotheses H_{0p_d}, H_{0p_c}, *and* H_{0f_d}.

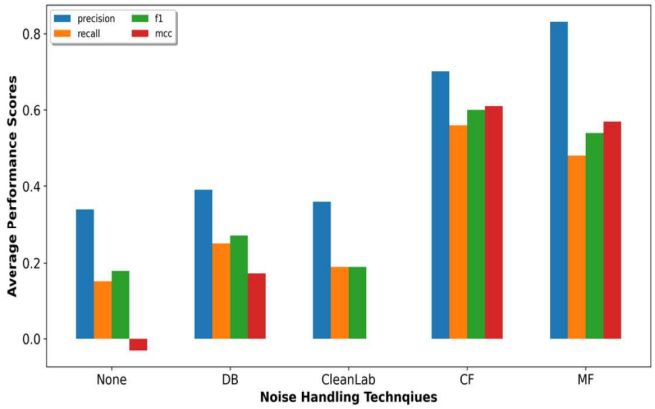


Fig. 11. Mean Precision, Recall, F1, and MCC after using each noise-handling technique on code review data.

To summarize, in the context of build outcome prediction, both MF and CleanLab demonstrate a statistically significant positive impact in terms of Recall, Precision, F1, and MCC. Among the two techniques, MF has a high positive impact, while CleanLab's impact, though positive, is modest. CF also shows a statistically significant improvement, suggesting that it improves the model's ability to predict true positives and true negatives, albeit it does not significantly impact the model's ability to avoid false positives. In contrast, DB yields a statistically significant increase in Recall but has a statistically significant negative effect on MCC, indicating a potential tradeoff between recall and overall predictive performance.

5.2 The Impact of Class Noise-handling on Predicting Negative Code Review Comments (RQ2)

To address the question "What is the impact of class noise-handling on predicting negative code review comments?", we structure our results into two analyses:

—Effect size of improvements per technique, Section 5.2.1.
—Significance analysis of the effects per technique, Section 5.3.

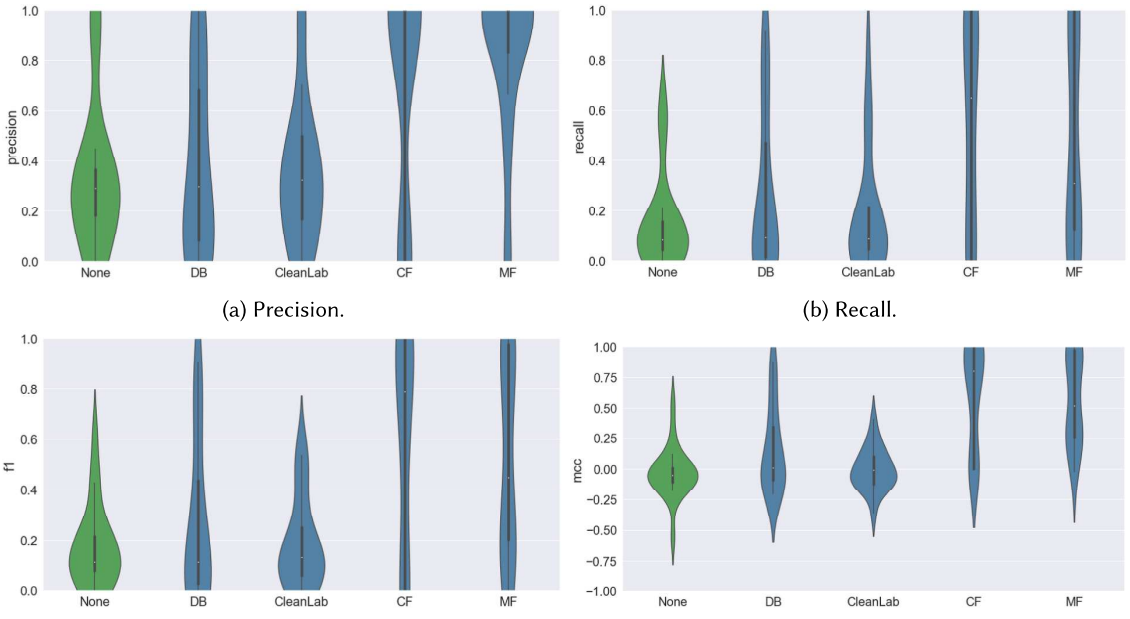
5.2.1 Descriptive Statistics. To evaluate the impact of the four noise-handling techniques on the performance of a model for predicting negative code review comments, we calculate the descriptive statistics of the four dependent variables after training on the control group data and each treatment level data respectively. Figure 11 is a bar plot that visualizes the mean percentages of Precision, Recall, F1, and MCC variables. The bar plot illustrates an improvement in the four variables. Specifically:

Applying the four class noise-handling techniques on the control group data had the following impact on Precision, on average:

—MF resulted in improving Precision from 34% to 83%.
—CF resulted in improving Precision from 34% to 70%.
—DB resulted in improving Precision from 34% to 39%.
—CleanLab resulted in improving Precision from 34% to 36%.

Applying the four class noise-handling techniques on the control group data had the following impact on Recall, on average:

—MF resulted in improving Recall from 15% to 48%.



(a) Precision.

(b) Recall.

(c) F1.

(d) MCC.

Fig. 12. Distribution of dependent variables for the four noise-handling techniques.

—CF resulted in improving Recall from 15% to 56%.
—DB resulted in improving Recall from 15% to 25%.
—CleanLab resulted in improving Recall from 15% to 19%.

Applying the four class noise-handling techniques on the control group data had the following impact on F1, on average:

—MF resulted in improving F1 from 18% to 54%.
—CF resulted in improving F1 from 18% to 60%.
—DB resulted in improving F1 from 18% to 27%.
—CleanLab resulted in improving F1 from 18% to 19%.

Applying the four class noise-handling techniques on the control group data had the following impact on MCC, on average:

—MF resulted in improving MCC from −0.03 to 0.57.
—CF resulted in improving MCC from −0.03 to 0.61.
—DB resulted in improving MCC from −0.03 to 0.17.
—CleanLab resulted in improving MCC from −0.03 to 0.001.

To better understand the impact of each class noise-handling technique, we visualized the distribution of each dependent variable before and after applying the treatment levels to the control group data. Figure 12(a)–(d) presents violin plots illustrating the distribution of Precision, Recall, F1 score, and MCC across all treatment levels. From these distributions, we observe the following:

—Applying any of the treatments to the training dataset increases the dispersion of the Precision and Recall metrics achieved by the RF model, indicating greater inconsistency in the model's generalization performance. This variability may be attributed to overfitting in certain train/validation splits.

Table 13. Descriptive Statistics of the Dependent Variables for an RF Model before and after Applying the Treatment Levels

			Precision		Recall		F1		MCC	
Noise algorithm	**Project**	N	Mean	SD	Mean	SD	Mean	SD	Mean	SD
None	googlesources	10	0.41	0.41	0.2	0.26	0.2	0.22	0.01	0.28
	wireshark	10	0.27	0.1	0.11	0.05	0.15	0.06	−0.07	0.08
MF	googlesources	10	0.99	0.04	0.83	0.31	0.86	0.28	0.87	0.24
	wireshark	10	0.67	0.41	0.14	0.13	0.22	0.18	0.27	0.2
CF	googlesources	10	1.0	0.0	0.94	0.12	0.96	0.07	0.96	0.07
	wireshark	10	0.4	0.52	0.18	0.25	0.25	0.33	0.27	0.35
DB	googlesources	10	0.57	0.4	0.46	0.36	0.47	0.35	0.35	0.41
	wireshark	10	0.22	0.26	0.05	0.08	0.08	0.12	−0.02	0.15
CleanLab	googlesources	10	0.36	0.31	0.20	0.25	0.20	0.21	0.012	0.21
	wireshark	10	0.35	0.27	0.18	0.25	0.18	0.15	−0.01	0.14

— Among all treatments, MF and CF result in the least dispersion in Precision and Recall, suggesting more stable performance compared to CleanLab and DB.
— CleanLab yields the lowest dispersion in F1 and MCC, indicating more consistent predictive behavior with respect to these metrics compared to the other treatment levels.

Table 13 shows the descriptive statistics of the dependent variables before and after applying the treatment. The table describes the mean and SD of the dependent variables across the 10 folds for every study subject. The data in the table shows that both MF and CF improved the four dependent variables in both study subjects, whereas DB and CleanLab improved the four dependent variables in one study subject, and slightly reduced them in the other subject.

Among the four class noise-handling techniques evaluated, MF and CF demonstrated the most substantial and consistent improvements across all performance metrics. In contrast, DB and CleanLab showed only marginal gains. Though CleanLab yielded low dispersion in F1 and MCC, its overall impact on performance was minimal. Overall, MF and CF are the most effective and reliable techniques for improving model performance in predicting negative code review comments.

5.3 Hypotheses Testing

To evaluate the hypotheses raised for RQ2, we began by running a Shapiro-Wilk analysis test to check the assumption of normality for the distribution of the dependent variables. Table 14 summarizes the normality test results for the four dependent variables.

Since we have issues in normality for the four dependent variables (p-value < 0.05) for the majority of dependent variables, we decided to compare the differences between each dependent variable under the four treatment levels and the control group using a non-parametric statistical analysis test.

Table 15 summarizes the results of the Kruskal-Wallis statistical test used to compare the four dependent variables across the control group and the four treatment conditions. The analysis reveals statistically significant differences ($p < 0.05$) in Precision, Recall, and MCC between the control group

Table 14. The Shapiro-Wilk Analysis Results for Each Dependent Variable after Applying the Noise-handling Techniques to the Code Review Data

Dependent variables	**Control group**	**DB**	**CleanLab**	**CF**	**MF**
Precision	Stats = 0.82, p < 0.05	Stats = 0.86, p < 0.05	Stats = 0.90, p < 0.05	Stats = 0.58, p < 0.05	Stats = 0.59, p < 0.05
Recall	Stats = 0.72, p < 0.05	Stats = 0.76, p < 0.05	Stats = 0.73, p < 0.05	Stats = 0.79, p < 0.05	Stats = 0.79, p < 0.05
F1	Stats = 0.86, p < 0.05	Stats = 0.8, p < 0.05	Stats = 0.86, p < 0.05	Stats = 0.76, p < 0.05	Stats = 0.82, p < 0.05
MCC	Stats = 0.87, p < 0.05	Stats = 0.85, p < 0.05	Stats = 0.96, p < .64	Stats = 0.75, p < 0.05	Stats = 0.86, p < 0.05

Table 15. Statistical Comparison Results between the Dependent Variables after Applying the Treatment Levels on the Code Review Data

	Precision	**Recall**	**F1**	**MCC**
Kruskal-Wallis H	Stats = 16.9	Stats = 8.9	Stats = 11.47	Stats = 30.2
Sig.	p < 0.05	p < 0.05	p < 0.06	p < 0.05

Table 16. Pairwise Comparison between the Dependent Variables for Each Treatment Level and the Control Group Using the Mann-Whitney U Test

	DB	**CleanLab**	**CF**	**MF**
Precision	Stats = 206.5, p < 0.87	Stats = 183, p < 0.65	Stats = 124, p < 0.05	Stats = 62, p < 0.05
Recall	Stats = 199.5, p < 0.05	Stats = 197.5, p < 0.96	Stats = 119, p < 0.05	Stats = 109, p < 0.05
F1	Stats = 194.5, p < 0.89	Stats = 197, p < 0.95	Stats = 113, p < 0.05	Stats = 82.5, p < 0.05
MCC	Stats = 151.5, p < 0.19	Stats = 181.5, p < 0.63	Stats = 45.0, p < 0.05	Stats = 27.5, p < 0.05

and the treatment groups. This suggests that applying the class noise-handling techniques has a significant impact on model performance. Notably, no statistically significant difference was observed in F1 scores across the groups, suggesting that F1 may be less sensitive to the applied treatments.

Table 16 presents the results of the Mann-Whitney U test used to perform pairwise comparisons between each treatment group and the control group across the four dependent variables.

The results indicate that both MF and CF treatments led to statistically significant improvements ($p < 0.05$) in all four dependent variables when compared to the control group. This suggests that these two techniques consistently improve the model's ability to predict negative code review comments.

In the case of DB, we only observed a statistically significant difference in Recall (Stats = 199.5, $p < 0.05$), while no significant differences were found in Precision, F1, or MCC ($p > 0.05$). CleanLab showed no statistically significant differences across any of the metrics when compared to the control group.

Table 17. The Descriptive Statistics of the Dependent Variables after Applying the Noise-handling Algorithms and the Two Feature Extraction Algorithms

	Noise algorithm	Precision		Recall		F1		MCC	
		Mean	SD	Mean	SD	Mean	SD	Mean	SD
BoW	None	0.9	0.07	0.9	0.06	0.9	0.06	0.3	0.28
	DB	0.9	0.16	0.81	0.28	0.84	0.3	0.19	0.13
	CF	0.91	0.08	**0.97**	0.15	0.94	0.10	0.46	0.15
	MF	**0.94**	0.21	**0.97**	0.35	**0.96**	0.3	**0.7**	0.31
CodeBERT	None	0.91	0.08	**0.96**	0.06	0.93	0.06	0.37	0.31
	DB	0.89	0.07	0.84	0.13	0.85	0.08	0.38	0.34
	CF	0.91	0.08	0.91	0.11	0.91	0.07	0.37	0.3
	MF	**0.95**	0.05	**0.96**	0.06	**0.96**	0.04	**0.61**	0.33

These results suggest that there is statistical evidence to support the rejection of the null hypotheses H_{0p_m}, H_{0p_c}, H_{0r_m}, H_{0r_c}, H_{0r_d}, H_{0f_m}, H_{f_c}, H_{0m_m}, *and* H_{0m_c}. Conversely, we did not find evidence to support the rejection of the hypotheses H_{0p_d}, H_{0p_cl}, H_{0r_cl}, H_{0f_d}, H_{0f_cl}, H_{0m_d}, *and* H_{0m_cl}.

To summarize, the results demonstrate that MF and CF significantly improve all performance metrics, validating their effectiveness in improving model performance. In contrast, DB has a limited impact, improving only Recall, and CleanLab shows no statistically significant impact on any dependent variable. This finding supports the superiority of removal-based techniques (i.e., MF and CF) over corrective-based techniques in the context of CI.

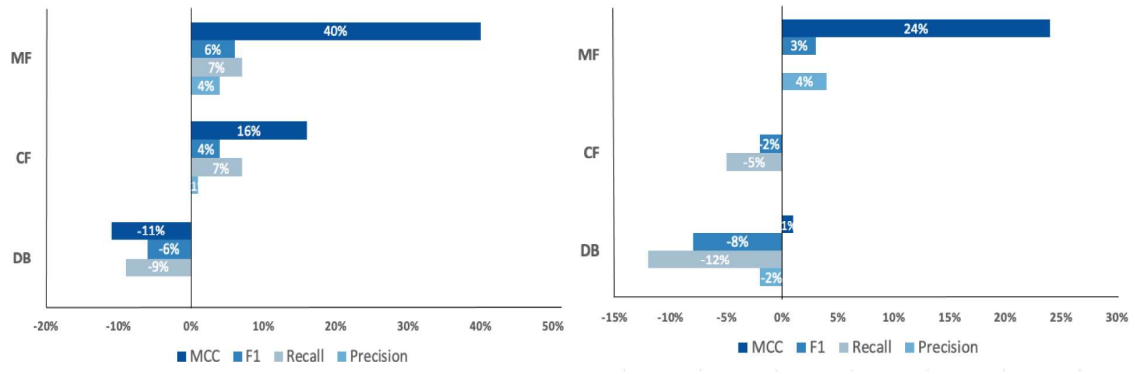
5.4 The Extent to Which BoW and CodeBERT Impact the Performance of Noise-handling Techniques (RQ3)

To examine the extent to which BoW and CodeBERT impact the performance of noise-handling techniques, we compare the predictive performance of RF models trained on both types of feature vectors respectively. The comparison was performed both before and after applying the DB, CF, and MF techniques.

Descriptive statistics of the model's performance in terms of Precision, Recall, F1, and MCC, are presented in Table 17. The highest mean performance results are highlighted in bold. As shown in the table, the mean performance scores of a model trained on BoW and CodeBERT are similar across the three noise-handling techniques. In other words, applying the three class noise-handling techniques to either BoW or CodeBERT feature vectors yields similar prediction results for build execution outcomes. For a more detailed overview of the mean performance scores of the model for each project, please refer to Tables A8 and A9.

Consistent with the findings in RQ2, we observe that applying MF to both BoW and CodeBERT feature vectors has the highest positive impact on performance (MCC with BoW = 0.7, MCC with CodeBERT = 0.61). Conversely, we observe that applying CF to CodeBERT feature vectors does not impact the model in terms of MCC (MCC remains at 0.37 before and after applying MF). This is contrary to the results that we observed when applying CF to the BoW feature vectors, which improved MCC from 0.3 to 0.46.

Figure 13(a) and (b) is diverging plots that show performance improvements of the model after applying the three class noise-handling techniques to BoW and CodeBERT feature vectors. The



(a) With BoW (b) With CodeBERT

Fig. 13. Performance improvements after applying class noise-handling to BoW and CodeBERT feature vectors.

improvements are measured relative to the performance scores achieved before applying any class noise-handling techniques to the data.

By examining the improvements in Figure 13(a), we observe that using DB has a negative impact on the predictive performance with respect to Recall, F1, and MCC, whereas Precision remained unchanged. On the other hand, we noticed that applying MF and CF to BoW feature vectors has the highest positive impact on the four performance measures (MCC improved by 40%, F1 improved by 6%, Recall improved by 7%, and Precision improved by 4%). Similarly, applying CF improved MCC by 16%, F1 by 4%, Recall by 7%, and Precision by 1%.

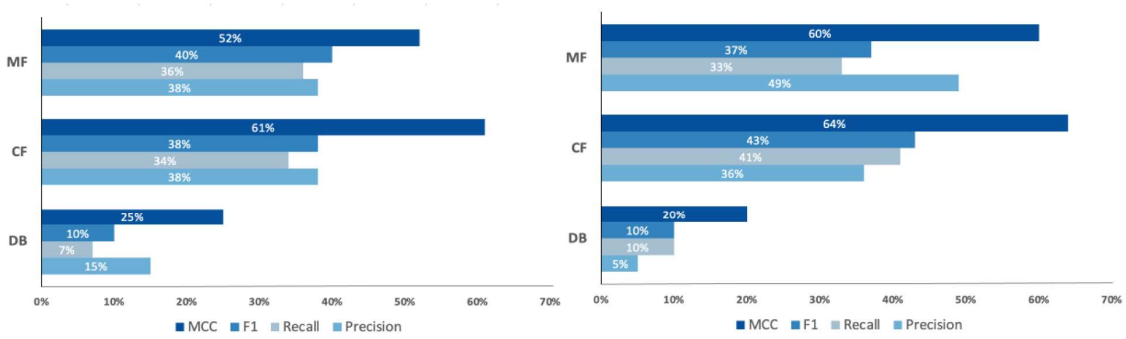
Similarly, the performance improvements depicted in Figure 13(b) suggest that applying DB to CodeBERT feature vectors has a negative impact on Precision, Recall, and F1, whereas MCC was improved by 1%. In addition, Precision and F1 improved after applying MF to CodeBERT feature vectors. Particularly, applying MF improved MCC by 24%, F1 by 3%, and Precision by 4%. In contrast to the impact of BoW on the performance of CF, the performance improvement scores illustrated in Figure 13(b) suggest that applying CF to CodeBERT feature vectors has a negative impact on F1 and Recall, whereas MCC and Precision remained unchanged. Particularly, Recall decreased by 5% and F1 decreased by 2%.

5.4.1 Code Change Request Prediction. To further validate our findings in the extent to which BoW and CodeBERT impact the performance of class noise-handling in the context of CI, we extended our analysis to include the two datasets used for addressing RQ2. Particularly, we compared the predictive performance of the model for negative code review prediction, both before and after applying the class noise-handling techniques to BoW and CodeBERT feature vectors.

Table 18 summarizes the descriptive statistics of the model's performance in predicting negative code review comments. The highest mean performance scores are highlighted in bold. As shown in Table 18, models trained on BoW and CodeBERT have a similar predictive performance across the three noise-handling techniques. However, contrary to the results reported in Table 17, we observe that applying CF to CodeBERT feature vectors leads to the highest Recall, F1, and MCC scores (Recall = 0.56, F1 = 0.60, and MCC = 0.61). On the other hand, the highest Precision was observed when applying MF to the CodeBERT feature vectors (Precision = 0.83). While the model's predictive performance has improved after applying CF and MF to both BoW and CodeBERT feature vectors, applying CF to CodeBERT feature vectors showed a slightly higher improvement compared to the other noise-handling techniques. Another observation from Table 18 suggests that there is a

Table 18. The Descriptive Statistics of the Dependent Variables after Applying the Noise-handling Algorithms and the Two Feature Extraction Algorithms

	Noise algorithm	Precision		Recall		F1		MCC	
		Mean	SD	Mean	SD	Mean	SD	Mean	SD
BoW	None	0.27	0.26	0.14	0.18	0.15	0.16	−0.07	0.19
	DB	0.42	0.39	0.21	0.28	0.25	0.32	0.18	0.34
	CF	0.65	0.49	0.48	0.43	0.53	0.44	0.54	0.44
	MF	0.65	0.37	0.50	0.37	0.55	0.36	0.45	0.44
CodeBERT	None	0.34	0.3	0.15	0.19	0.17	0.16	−0.03	0.20
	DB	0.39	0.38	0.25	0.33	0.27	0.32	0.17	0.35
	CF	0.70	0.47	**0.56**	0.43	**0.60**	0.43	**0.61**	0.43
	MF	**0.83**	0.33	0.48	0.42	0.54	0.40	0.57	0.38



(a) With BoW

(b) With CodeBERT

Fig. 14. Performance improvements after applying class noise-handling to BoW and CodeBERT feature vectors.

disparity in the distribution of the performance scores, with high SD in some cases. This indicates that the performance scores can deviate substantially from the mean for some projects.

Figure 14(a) and (b) shows the performance improvements of the model for negative code review prediction after applying the three class noise-handling techniques to BoW and CodeBERT feature vectors. The improvements are measured relative to the performance scores achieved before applying any class noise-handling techniques to the data.

By examining the performance improvements in Figure 14(a) and (b), we observe that applying any class noise-handling techniques to BoW and CodeBERT feature vectors has a positive impact on the predictive performance of a model for negative code review comment prediction. Particularly, using CF and MF for noise-handling in CodeBERT feature vectors demonstrates the highest improvement to MCC (MCC improved by 60% with MF, and MCC improved by 64% with CF). Conversely, applying DB to BoW feature vectors demonstrates the highest improvement with respect to MCC and Precision (MCC improved by 25%, and Precision improved by 15%).

In general, using BoW or CodeBERT has a similar impact on the performance of the three class noise-handling techniques in the context of build outcome and negative code review comment predictions.

5.5 The Extent to Which Hyperparameter Tuning Impacts the Model Performance for Predicting the Outcome of Build Execution (RQ4)

To answer the RQ of whether tuning hyperparameters can improve the predictive performance of the model for build outcome predictions, we examined the improvement in performance provided by the Grid Search algorithm on the RF model before and after applying each treatment. The Grid Search algorithm was configured to explore 216 different hyperparameter combinations for each treatment, using the hyperparameters listed in Table 8.

Table 19 presents the average performance metrics—Precision, Recall, F1, and MCC—recorded across 14 study subjects, comparing model performance before tuning (Default column) and after tuning (Tuned column). The observed changes are summarized under the Change column. Overall, the results suggest that hyperparameter tuning leads to a generally modest improvement in model performance. These improvements are most notable with respect to the Recall and MCC metrics, which reflect an improved ability of the model's to correctly predict failing builds.

The most significant gains in Recall and MCC were observed when tuning was applied before any treatment, and also after applying CleanLab, with increases reaching up to 0.33 in Recall and 0.28 in MCC. These findings indicate that tuning can make the model more effective at predicting build outcomes, especially in the absence of noise-handling treatments. The same observation can be made when CleanLab is used for curating the training dataset. Interestingly, while tuning before treatment generally improves performance, the application of treatments without tuning resulted in even higher gains in some cases. This suggests that the impact of tuning depends on the configured model's hyperparameters and the characteristics of the data, with some treatments inherently improving the model's robustness more than tuning alone.

Precision, however, remains relatively stable or slightly decreases with tuning, indicating that while the model becomes better at identifying failures, it does not necessarily improve in avoiding false positives. In a CI context, this tradeoff is often acceptable, particularly when early detection of true build failures is more critical than minimizing occasional false alarms. As such, the benefit of hyperparameter tuning is more pronounced in scenarios where recall is prioritized over precision.

These trends are further supported by the boxplots presented in Figure 15, which illustrate the distribution of performance metrics under both default and tuned conditions across the different treatment levels. In the baseline setting before any noise-handling is applied (Figure 15(a)), hyperparameter tuning leads to clear improvements in Recall and MCC, confirming that tuning alone can help the model better identify failed builds. After applying DB (Figure 15(b)), the impact of tuning is less paramount but still positive, especially for Recall, though with minimal change in MCC.

The impact of tuning is particularly strong after applying CleanLab (Figure 15(c)), where improvements in MCC is visible. This suggests that CleanLab and tuning work well together to improve the model's overall prediction quality. After applying CF (Figure 15(d)), tuning has a moderate effect, with stable Precision and F1, and only minor improvements in Recall and MCC. Interestingly, after applying MF (Figure 15(e)), tuning leads to a decrease in MCC, despite high Recall and F1 scores. This suggests that when MF is applied, tuning may inadvertently lead to overfitting, likely due to increased class imbalance introduced during filtering, which can skew the model's learning and reduce generalization.

In summary, the results conclude that hyperparameter tuning is most effective when applied before noise-handling or in combination with CleanLab. While its impact varies across treatments, tuning generally improves the model's ability to predict failing builds as demonstrated in the gains achieved with respect to Recall and MCC metrics.

Table 19. Mean Performance Metrics Before and After Hyperparameter Tuning

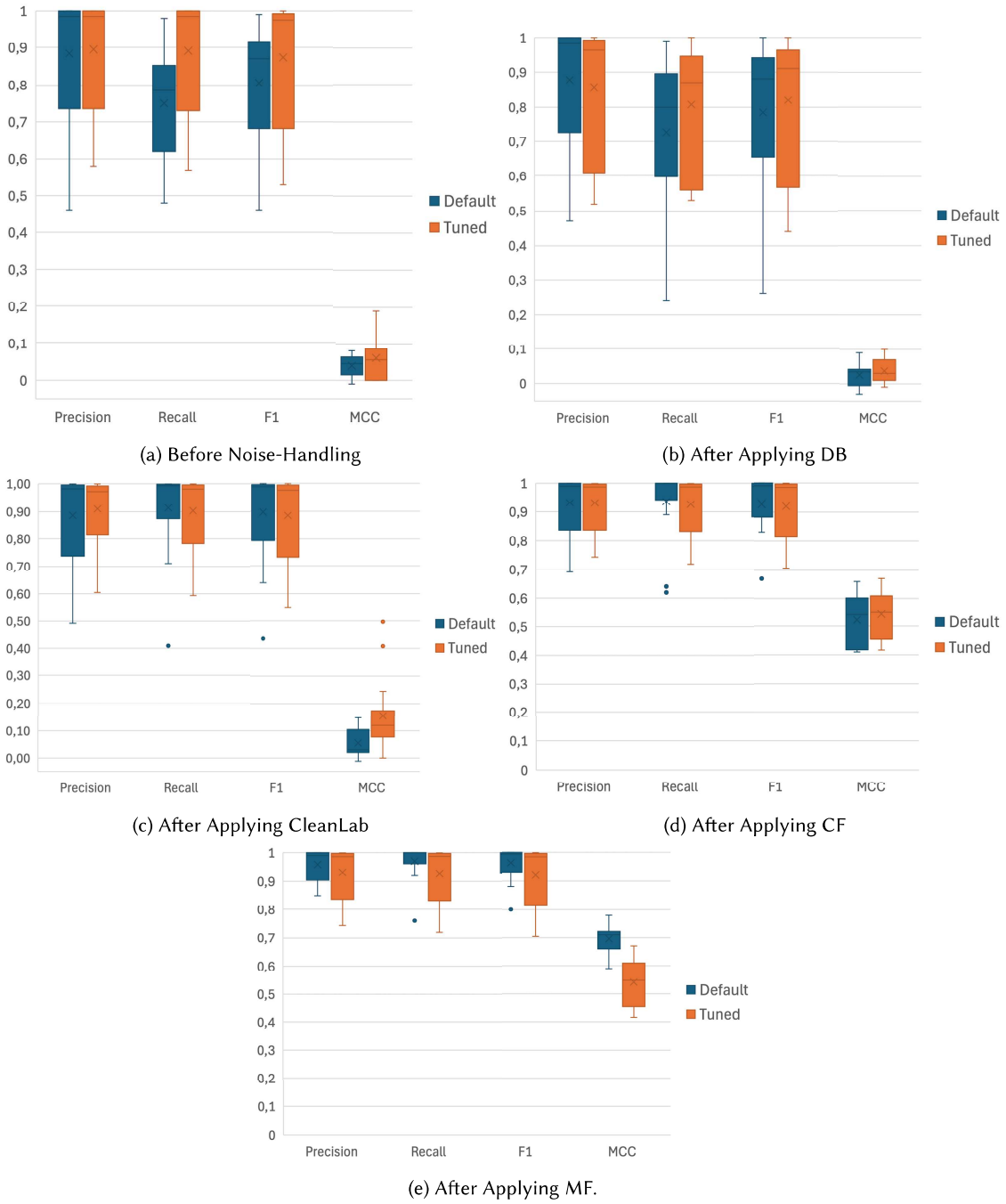
		None			DB			CF			MF			CleanLab		
Project	Metric	Default	Tuned	Change	Default	Tuned	Change	Default	Tuned	Change	Default	Tuned	Change	Default	Tuned	Change
querydsl	Precision	0.98	0.98	0	0.97	0.95	−0.02	0.98	0.98	0	0.99	0.98	−0.01	0.98	0.97	−0.01
	Recall	0.75	0.98	0.23	0.75	0.87	0.12	1	0.99	−0.01	1	0.99	−0.01	1	0.98	−0.02
	F1	0.85	0.97	0.12	0.84	0.91	0.07	0.99	0.98	−0.01	0.99	0.98	−0.01	0.99	0.97	−0.02
	MCC	0.05	0.19	0.14	−0.02	0.01	0.03	0.59	0.60	0.01	0.71	0.60	−0.11	0.13	0.24	0.11
restlet-framework	Precision	0.57	0.6	0.03	0.52	0.54	0.02	0.8	0.76	−0.04	0.85	0.76	−0.09	0.59	0.76	0.17
	Recall	0.54	0.59	0.05	0.24	0.53	0.29	0.64	0.72	0.08	0.92	0.72	−0.20	0.71	0.68	−0.03
	F1	0.55	0.55	0	0.26	0.44	0.18	0.67	0.70	0.03	0.88	0.70	−0.18	0.64	0.64	0
	MCC	0.06	0.15	0.09	0.09	0.1	0.01	0.42	0.47	0.05	0.71	0.47	−0.24	0.13	0.41	0.27
robospice	Precision	0.46	0.58	0.12	0.47	0.52	0.05	0.69	0.74	0.05	0.85	0.74	−0.11	0.49	0.60	0.11
	Recall	0.48	0.57	0.09	0.26	0.54	0.28	0.62	0.73	0.11	0.76	0.73	−0.03	0.41	0.59	0.18
	F1	0.46	0.53	0.07	0.32	0.51	0.19	0.64	0.72	0.08	0.8	0.72	−0.08	0.44	0.55	0.11
	MCC	0.04	0.1	0.06	0.02	0.02	0	0.41	0.45	0.04	0.67	0.45	−0.22	0.07	0.15	0.08
rultor	Precision	0.72	0.72	0	0.69	0.58	−0.11	0.8	0.78	−0.02	0.86	0.78	−0.08	0.71	0.82	0.11
	Recall	0.62	0.7	0.08	0.5	0.56	0.06	0.89	0.78	−0.11	0.94	0.78	−0.16	0.88	0.79	−0.09
	F1	0.66	0.66	0	0.58	0.56	−0.02	0.83	0.75	−0.08	0.9	0.75	−0.15	0.78	0.76	−0.02
	MCC	0.07	0.08	0.01	0	0.01	0.01	0.41	0.44	0.03	0.63	0.44	−0.19	0.09	0.50	0.40
spark	Precision	0.99	0.99	0	0.99	0.98	−0.01	0.99	0.99	0	0.99	0.99	0	0.99	0.98	−0.01
	Recall	0.83	0.99	0.16	0.82	0.86	0.04	1	0.99	−0.01	1	0.99	−0.01	0.99	0.98	−0.01
	F1	0.9	0.98	0.08	0.89	0.91	0.02	0.99	0.99	0	1	0.99	−0.01	0.99	0.98	−0.01
	MCC	0.08	0.08	0	0.05	0.09	0.04	0.42	0.45	0.03	0.71	0.45	−0.26	0.03	0.14	0.12
qulice	Precision	0.98	0.98	0	0.98	0.96	−0.02	0.99	0.99	0	0.99	0.99	0	0.98	0.96	−0.01
	Recall	0.8	0.98	0.18	0.78	0.78	0	1	0.99	−0.01	1	0.99	−0.01	0.99	0.98	−0.01
	F1	0.88	0.97	0.09	0.87	0.86	−0.01	0.99	0.98	−0.01	0.99	0.98	−0.01	0.98	0.97	−0.02
	MCC	−0.01	0.06	0.07	−0.03	0	0.03	0.61	0.64	0.03	0.72	0.64	−0.08	−0.01	0.08	0.09
thredds	Precision	0.74	0.74	0	0.74	0.62	−0.12	0.85	0.86	0.01	0.92	0.86	−0.06	0.75	0.79	0.04
	Recall	0.64	0.74	0.1	0.63	0.56	−0.07	0.96	0.85	−0.11	0.97	0.85	−0.12	0.86	0.75	−0.11
	F1	0.69	0.69	0	0.68	0.57	−0.11	0.9	0.84	−0.06	0.94	0.84	−0.10	0.80	0.65	−0.15
	MCC	0	0.04	0.04	−0.02	−0.01	0.01	0.56	0.58	0.02	0.76	0.58	−0.18	0.03	0.10	0.07
rxjava-jdbc	Precision	1	1	0	1	1	0	1	1	0	1	1	0	1	1	0
	Recall	0.84	1	0.16	0.87	0.91	0.04	1	1	0	1	1	0	0.99	1	0
	F1	0.91	1	0.09	0.93	0.95	0.02	1	1	0	1	1	0	1	1	0
	MCC	0.03	0	−0.03	0.04	0.04	0	0.52	0.52	0	0.72	0.52	−0.20	0.06	0.12	0.07
sms-backup-plus	Precision	0.99	0.99	0	0.99	0.97	−0.02	0.99	0.99	0	1	0.99	−0.01	0.99	0.97	−0.01
	Recall	0.77	0.99	0.22	0.95	0.94	−0.01	1	0.99	−0.01	1	0.99	−0.01	0.99	0.99	0
	F1	0.86	0.98	0.12	0.97	0.96	−0.01	1	0.99	−0.01	1	0.99	−0.01	0.99	0.98	−0.01
	MCC	0	0	0	0.01	0.02	0.01	0.66	0.67	0.01	0.78	0.67	−0.11	−0.01	0	0.01
spring-cloud	Precision	0.95	0.95	0	0.95	0.9	−0.05	0.96	0.96	0	0.98	0.96	−0.02	0.95	0.91	−0.04
	Recall	0.62	0.95	0.33	0.72	0.87	0.15	1	0.96	−0.04	1	0.96	−0.04	0.98	0.91	−0.07
	F1	0.74	0.92	0.18	0.82	0.88	0.06	0.98	0.95	−0.03	0.99	0.95	−0.04	0.96	0.91	−0.05
	MCC	0.02	0	−0.02	0.04	0.04	0	0.51	0.53	0.02	0.72	0.53	−0.19	0.03	0.07	0.04
traccar	Precision	1	1	0	1	1	0	1	1	0	1	1	0	1	1	0
	Recall	0.84	1	0.16	0.89	1	0.11	1	1	0	1	1	0	1	1	0
	F1	0.91	1	0.09	0.94	1	0.06	1	1	0	1	1	0	1	1	0
	MCC	0.05	0.05	0	0.04	0.07	0.03	0.41	0.42	0.01	0.59	0.42	−0.17	0.03	0.09	0.06
webcam-capture	Precision	1	1	0	0.99	0.99	0	1	1	0	1	1	0	0.99	0.99	−0.01
	Recall	0.89	1	0.11	0.89	0.93	0.04	1	1	0	1	1	0	1	0.99	−0.01
	F1	0.94	0.99	0.05	0.94	0.96	0.02	1	0.99	−0.01	1	0.99	−0.01	1	0.99	−0.01
	MCC	0.07	0	−0.07	0.07	0.01	−0.06	0.56	0.57	0.01	0.7	0.57	−0.13	0.15	0.15	0
wire	Precision	1	1	0	1	1	0	1	1	0	1	1	0	1	1	0
	Recall	0.98	0.99	0.01	0.99	0.99	0	1	1	0	1	1	0	1	1	0
	F1	0.99	1	0.01	1	1	0	1	1	0	1	1	0	1	1	0
	MCC	0.03	0.03	0	0.04	0.05	0.01	0.6	0.60	0	0.6	0.60	0	0.03	0.12	0.09
yobi	Precision	1	1	0	1	0.99	−0.01	1	1	0	1	1	0	1	0.99	0
	Recall	0.9	1	0.1	0.91	0.97	0.06	1	1	0	1	1	0	1	1	0
	F1	0.94	0.99	0.05	0.95	0.98	0.03	1	1	0	1	1	0	1	0.99	0
	MCC	0.06	0.06	0	0.03	0.07	0.04	0.62	0.63	0.01	0.73	0.63	−0.10	0	0	0

6 Discussion

In this section, we answer the two RQs and provide insight into the characteristics of lines of code that were identified as noisy by the three examined class noise-handling techniques.

6.1 RQ1: What is the Impact of Applying Class Noise-handling Techniques on Predicting the Outcome of Builds in CI?

The evaluation results of this study reveal that applying removal-based techniques to training data has a consistent positive impact on the predictive performance of the ML model to build outcome predictions. Specifically, the more conservative technique, known as the MF, was found to be more effective compared to the aggressive technique, known as the CF. This suggests that by employing



(a) Before Noise-Handling

(b) After Applying DB

(c) After Applying CleanLab

(d) After Applying CF

(e) After Applying MF.

Fig. 15. Mean performance scores before and after tuning RF hyperparameters.

techniques that preserve a larger portion of the data, the model benefits from a more comprehensive analysis of the underlying patterns and trends, resulting in better predictions. In addition, the evaluation showed that leaving the noise intact and relying on the tolerance capability of the model leads to a higher predictive performance of build outcomes compared to when applying the DB technique to the training data.

These findings align with previous studies that have advocated for the use of conservative noise removal techniques to improve the performance of ML models [16]. By employing techniques that preserve a larger portion of the data, the model benefits from a more comprehensive understanding of the underlying patterns and trends, resulting in improved predictions.

6.2 RQ2: What is the Impact of Applying Class Noise-handling Techniques on Predicting Negative Code Review Comments?

In line with the evaluation results of RQ1, our findings reveal that the removal-based techniques consistently improve the predictive performance of an ML model for change request predictions. Additionally, we observed that the DB technique improves the overall performance of the model, albeit to a lower extent than the statistical techniques. Notably, the improvements achieved by applying the domain-knowledge technique were in the prediction of comments that requested a code change, rather than comments that are approved for integration. This was materialized in the significant improvements gained in MCC, but not F1.

Previous studies that examined the use of sentiment analysis tools in SE contexts have commonly employed tools such as SentiStrength [22, 53] or NLTK [44]. One study conducted by Jongeling et al. [25] found that these tools not only led to poor accuracies but also resulted in significant disagreements in classifications. Hence, applying noise-handling techniques to the training data produced by these sentiment analysis tools can potentially reduce the contradictory labeling (improve accuracy) and consequently can improve the predictive performance of the final ML models.

Finally, it is important to interpret the evaluation results for both RQ1 and RQ2 in light of the noise ratios present in the data. In this study, the highest observed noise ratio among all analyzed projects was below 40%. This implies that the findings may vary if the ratio of class noise exceeds this threshold. According to Teng [52], when applying noise-handling techniques to data with class noise exceeding 40%, the improvement in accuracy becomes inconsistent. Therefore, it is crucial to consider the level of noise ratio in the training data when applying these techniques in practice.

6.3 RQ3: To Which Extent Do BoW and CodeBERT Feature Extraction Techniques Impact the Performance of the Noise-handling Techniques?

Our analysis results reveal a similar effect on model's predictive performance for build outcome predictions when applying the three noise-handling techniques on BoW and CodeBERT features. On the one hand, the predictive performance of the model improved when applying MF and CF. On the other hand, performance decreased when applying DB to the training data.

In the context of negative code review prediction, our results show that applying any class noise-handling techniques to the extracted features with BoW or CodeBERT has a positive impact on the predictive performance of a model. While the improvement trends are similar across the two feature extraction algorithms, it is important to note that the model's predictive performance was most improved when applying the CF and MF algorithms to the training data.

6.4 RQ4: To Which Extent Does Model Hyperparameter Tuning Impact the Effectiveness of Predicting Build Outcomes before and after Handling Class Noise?

The findings from the fourth experiment demonstrate that hyperparameter tuning has a generally modest impact on improving the predictive performance of models for build outcome prediction in the context of CI. By applying a Grid Search algorithm, we examined the impact of tuning both before and after applying four different noise-handling treatments across 14 diverse study subjects.

The results indicate that tuning is most effective when applied to the baseline model (i.e., without applying any noise-handling) or after applying CleanLab. In these two cases, notable gains were

Table 20. An Excerpt of Lines of Code Classified by the Examined Noise-handling Techniques

Contents	Actual class	MF	CF	DB	CleanLab
/*	1	kept	kept	kept	kept
import org.geolatte.geom.*;	1	kept	kept	kept	kept
* Licensed under the Apache License, Version 2.0 (the "License");	1	kept	kept	kept	kept
*/	1	kept	kept	kept	kept
public Collection<Predicate>string(StringExpression expr,	1	noisy	noisy	noisy	kept
StringExpression other,					
}	1	kept	noisy	kept	kept
return new BooleanExpression(underlyingMixin) {	1	noisy	noisy	noisy	kept
super(ExpressionUtils.path(type, metadata));	1	noisy	noisy	kept	kept
return UNSUPPORTED_OPERATORS.contains(((Operation<?>)	0	noisy	noisy	noisy	kept
filter).getOperator());					
listeners.preExecute(context);	1	noisy	noisy	kept	noisy
public static NumberExpression<Double>ymax(GeometryExpression<?>expr)	1	noisy	noisy	kept	kept
{					

observed in Recall and MCC—up to 0.33 and 0.28, respectively—highlighting improved ability to predict failing builds. Interestingly, while tuning generally improves predictive performance in these two cases, the impact is not uniformly positive across all treatment types. For example, after applying DB, tuning still yielded some improvement, mainly in Recall, but the gains were relatively modest and negative in other cases. Similarly, CF exhibited only minor improvements in MCC, with Precision and F1 remaining unchanged, indicating limited impact in predicting passing builds.

A more nuanced outcome was observed when tuning was examined for an impact after training on MF-curated datasets. There, tuning led to a reduction in performance across the examined performance metrics. This suggests a possibility of introducing overfitting as a consequence of increased class imbalance created during the filtering process. Since MF removes noisy instances, it may distort the class distribution in a way that causes the tuned model to perform well on the training set but generalize poorly on the validation set.

Precision, across all treatments and tuning conditions, remained relatively stable or decreased slightly. This indicates that while tuning helps in correctly predicting more failing builds (i.e., higher Recall), it does not necessarily improve the model's ability to avoid false positives.

It is important to note that no statistical tests were conducted to determine the significance of the observed performance differences. Therefore, while the trends are visually and numerically indicative, future work should validate the significance level of the findings.

In summary, while hyperparameter tuning provides modest benefits to model performance before applying treatments in some cases, its effectiveness is closely tied to the treatment applied to the training data. CleanLab appears to pair well with tuning by improving model recall, whereas MF may require more cautious tuning to avoid overfitting. In general, hyperparameter tuning for RF resulted in modest improvements when compared to the RF default values.

6.5 Characteristics of Noisy Lines of Code

To gain a better understanding of the nature of lines of code that are identified as noisy by the four examined noise-handling techniques, we observed patterns in a sample of the analyzed code and sought to identify syntactical characteristics of these lines.

Table 20 provides a sample of lines of code that we randomly selected and analyzed from the "querydsl" project. By examining the sample of lines, we observe the following characteristics among the sample of lines that are classified as noisy and non-noisy by the MF, CF, DB, and CleanLab.

Our observations for lines that were identified as noisy and non-noisy by the four noise-handling techniques reveals distinct patterns in their classification behaviors. Lines consistently identified as non-noisy (kept) by all techniques, such as comment delimiters (/*, */) and import statements (e.g., import org.geolatte.geom.*;), generally correspond to structural or documentation-related elements within the code. These lines are syntactically complete and unlikely to influence functional correctness, which explains the high agreement among the techniques in labeling them as non-noisy.

In contrast, lines involving control structures, such as public Collection<Predicate> string(String-Expression expr, StringExpression other, and return new BooleanExpression(underlyingMixin), exhibited greater disagreement across techniques. The MF, CF, and DB techniques frequently labeled these lines as noisy, whereas CleanLab tended to retain them. This difference suggests that MF, CF, and DB techniques are more sensitive to control structure statements, possibly interpreting them as indicators of labeling errors, while CleanLab, leveraging probabilistic learning, appear to treat these code constructs as non-noisy variations.

Moreover, complex expression lines, such as `return UNSUPPORTED_OPERATORS.contains (((Operation<?>) filter).getOperator());` were identified as noisy by MF, CF, and DB techniques, even though they exhibit syntactically valid functional logic. CleanLab, however, retained this line, highlighting its tolerance against variability in code constructs. A notable case of disagreement was observed in the line `listeners.preExecute(context);` where CleanLab also marked the line as noisy, aligning with the statistical techniques, suggesting that certain functional constructs may introduce uncertainty, and hence be treated as potential noise across different methods.

Overall, these observations indicate that statistical noise-handling techniques (Majority and CFs, Domain Knowledge) tend to favor simplicity when identifying non-noisy lines, while CleanLab demonstrates a higher tolerance for more complex and partially formed constructs. This difference underscores the importance of selecting a noise-handling strategy aligned with the type of data and the nature of the prediction task.

6.6 Confounding Factors

It is important to note that the effectiveness of the examined class noise-handling techniques is subject to several factors related to the pre-processing activities performed on the training data. Three key activities were identified as potential influencers: data collection, feature extraction, and class-balancing techniques.

Data Collection. as a crucial aspect of the study, relies on the reliability of the build records obtained from TravisTorrent. However, it is important to acknowledge that the accuracy of all the collected build job outcomes and commit hashes cannot be guaranteed. This introduces the possibility of inaccurate mappings between code changes and target class values within the dataset, potentially impacting the effectiveness of the noise-handling techniques under examination.

Feature Extraction. The choice of feature extraction technique can influence the predictive performance of the RF model in both contexts. The specific method used to measure the frequency of tokens in the input code, such as word embeddings or TF-IDF, and variations in the configuration of the BoW model, including n-gram settings, may have an impact on the performance of the four examined techniques. Exploring different feature extraction algorithms and BoW configuration parameters is a avenue for future research to better understand their effect on the effectiveness of the examined techniques.

Class Balancing. The selection of a class balancing technique can also affect the performance of noise-handling techniques. Different methods for balancing the classes should be empirically

investigated to assess their impact on the effectiveness of the four examined noise-handling techniques.

Considering these factors, it is crucial to interpret the study results with awareness of the potential influence that these pre-processing activities can have on the results. Thus, future research should focus on addressing these areas to strengthen the validity of the findings.

7 Research Validity

When analyzing the threats to the validity of our study, we follow the framework recommended by Wohlin et al. [57] and discuss the validity in terms of external, internal, construct, and conclusion.

7.1 External Validity Threats

External validity refers to the degree to which the results can be generalized outside the context of the current study.

Sample Size. The study subjects used in Experiment 2 belong to two projects only. Hence, it is difficult to know whether the results drawn from this experiment can be generalized to the overall population of projects. However, we increase the likelihood of generalizability by using two different data sources to collect our sample projects and by randomly selecting a sample of code comments for annotation.

Programming Languages. A potential threat to external validity in empirical SE research is the representativeness of the study subjects to other programming languages. However, we used a language-agnostic tool for extracting features from the study subjects. Hence, the probability that our findings apply to other programming languages in the two SE contexts increases.

7.2 Internal Validity Threats

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

Instrumentation. A potential internal threat is the presence of undetected issues in the scripts we implemented and used for collecting code reviews and build data. This threat was controlled by carrying out a careful inspection of the scripts and testing them on small subsets.

Reuse of Public Datasets. Since the analysis results of this study are based on public datasets, we can not rule out the possibility of encountering erroneous data entries in the collected build and code review comment datasets. However, we minimize this threat by manually validating a few of the collected review comments by our mining tools using the Gerrit web interface.

Subjectivity in Labeling. Given the subjective nature of annotating code review commits, there is a possibility of introducing inconsistency in the labeling of code commits, leading to class noise. We minimized this risk by calculating the interrater agreement among the three authors responsible for annotating a subset of 200 code commits. The analysis showed a high level of agreement, exceeding 95% for negative comments and 80% for positive comments.

7.3 Construct Validity Threats

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

Choice of the ML Model. This study employed a RF model as the final learner for evaluating the impact of class noise-handling techniques in both SE contexts. It is difficult to assert whether tuning the parameters of the model or using different types of models (e.g., convolutional NNs) would

change the predictive performance results. To minimize this threat, we compared the performance of three different models in predicting build outcomes. The comparison results showed that RF outperformed a two-dense layered NN and an XGBoost model. Therefore, we decided to use RF for measuring the effects of class noise-handling.

Data Balancing Technique. In this study, we used an oversampling technique to deal with the problem of imbalanced training data. Using other techniques such as down-sampling or hybrid ones may change the reported results and, thereby, the conclusions. However, we chose to use an oversampling technique based on the recommendations of Mendoza et al. [33], which suggests that using oversampling yields better results than down-sampling and hybrid techniques.

7.4 Conclusion Validity Threats

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

Class Noise Measurement. Our measurement of class noise is based on the ratio of contradictory entries in the data. However, inaccurate class values can also appear among entries that are not necessarily contradictory. This means that if we use different metrics for measuring class noise, we might reach different conclusions about the effect of each technique. However, our choice of using this metric for measuring class noise is motivated by our findings in previous research [4, 5], where we identified a large number of contradictory entries in a regression testing dataset.

Compatibility of Class Noise-handling Techniques. This study examined the impact of three class noise-handling techniques that handle class noise in different ways. That is, the removal-based techniques (MF and CF) utilize statistical measures to determine the accuracy of code labeling. In contrast, DB leverages domain knowledge regarding code changes to identify lines of code that require labeling. As a result, comparing the three techniques may pose challenges due to their inherent differences. Nonetheless, the objective of our study was to investigate how these diverse techniques impact the predictive performance of ML models when applied to software engineering data.

8 Conclusion and Future Work

In conclusion, our evaluation results reveal that applying removal-based class noise-handling techniques significantly improves the predictive performance of ML models in both build outcome and negative code review prediction tasks. Among the examined techniques, MF and CF consistently improved performance across most of the evaluation metrics. In build outcome predictions, MF showed higher effectiveness compared to CF, DB, and CleanLab, suggesting that conservative removal of suspected noisy entries yields better model performance. Conversely, for negative code review predictions, CF slightly outperformed MF, indicating that an aggressive agreement among classifiers can be beneficial in tasks characterized by subjective annotations.

Interestingly, the DB technique exhibited mixed results: it improved predictive performance in negative code review prediction but was less effective for build outcome prediction compared to leaving the noise intact. This indicates that domain-driven techniques may introduce biases when applied to certain types of CI-related prediction tasks.

In addition to these observations, our analysis revealed that CleanLab exhibited a distinct behavior compared to CF, MF, and DB. Particularly, CleanLab showed a conservative tendency in classifying lines as noisy, particularly retaining syntactically or semantically valid yet complex code constructs that other techniques tended to filter out. This suggests that CleanLab's probabilistic learning offers resilience against complex code patterns. However, its overall impact on predictive performance was more moderate compared to removal-based methods, particularly in build outcome predictions.

Hyperparameter tuning also proved crucial when used in isolation of noise-handling techniques or in combination with CleanLab. The results showed noticeable improvements in these two cases, as observed from the Recall and MCC scores. This highlights the importance of jointly tuning model paramters and applying noise-handling strategies to maximize the detection of fault-prone changes in the context of CI.

Several avenues for future work emerge from our study. First, expanding the sample size for negative code review prediction datasets would enhance the generalizability of noise-handling effectiveness in this context. Second, incorporating a broader range of class noise-handling techniques, including more noise-handling techniques would strengthen the external validity of the findings. Third, analyzing the behavior of noise-handling techniques under different class noise ratios, especially beyond the 40% threshold observed in this study, would provide practitioners with more tailored recommendations depending on project-specific noise levels. Finally, future studies should examine whether the size and complexity of software projects influence the effectiveness of noise-handling techniques, as larger projects may exhibit different noise patterns and model sensitivities compared to smaller ones.

References

[1] Toufique Ahmed, Amiangshu Bosu, Anindya Iqbal, and Shahram Rahimi. 2017. SentiCR: A customized sentiment analysis tool for code review interactions. In *Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 106–111.

[2] Khaled Al-Sabbagh, Miroslaw Staron, and Regina Hebig. 2022. Predicting build outcomes in continuous integration using textual analysis of source code commits. In *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*, 42–51.

[3] Khaled Walid Al-Sabbagh, Regina Hebig, and Miroslaw Staron. 2020. The effect of class noise on continuous test case selection: A controlled experiment on industrial data. In *Proceedings of the International Conference on Product-Focused Software Process Improvement*. Springer, 287–303.

[4] Khaled Walid Al-Sabbagh, Miroslaw Staron, and Regina Hebig. 2022. Improving test case selection by handling class and attribute noise. *Journal of Systems and Software* 183 (2022), 111093.

[5] Khaled Walid Al-Sabbagh, Miroslaw Staron, Regina Hebig, and Wilhelm Meding. 2020. Improving data quality for regression test selection by reducing annotation noise. In *Proceedings of the 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 191–194.

[6] Tanmay Basu. 2023. Identification of the relevance of comments in codes using bag of words and transformer based models. arXiv:2308.06144. Retrieved from https://arxiv.org/abs/2308.06144

[7] C. Bentéjac, A. Csörgo, and G. Martínez-Muñoz. 1911. A comparative analysis of XGBoost. Retrieved from https://arxiv.org/abs/1911.01914

[8] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research* 13, 1 (2012), 281–305.

[9] Carla E. Brodley and Mark A. Friedl. 1999. Identifying mislabeled training data. *Journal of Artificial Intelligence Research* 11 (1999), 131–167.

[10] Lin Chen, Bin Fang, Zhaowei Shang, and Yuanyan Tang. 2018. Tackling class overlap and imbalance problems in software defect prediction. *Software Quality Journal* 26 (2018), 97–125.

[11] Davide Chicco and Giuseppe Jurman. 2020. The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation. *BMC Genomics* 21, 1 (2020), 1–13.

[12] Francois Chollet. 2015. Keras. Retrieved from https://github.com/fchollet/keras

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of deep bidirectional transformers for language understanding. arXiv:1810.04805. Retrieved from https://arxiv.org/abs/1810.04805

[14] Ionut-Catalin Donca, Ovidiu Petru Stan, Marius Misaros, Dan Gota, and Liviu Miclea. 2022. Method for continuous integration and deployment using a pipeline generator for agile software projects. *Sensors* 22, 12 (2022), 4637.

[15] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A pre-trained model for programming and natural languages. arXiv:2002.08155. Retrieved from https://arxiv.org/abs/2002.08155

[16] Norman E. Fenton and Martin Neil. 1999. A critique of software defect prediction models. *IEEE Transactions on Software Engineering* 25, 5 (1999), 675–689.

[17] Keheliya Gallaba, Christian Macho, Martin Pinzger, and Shane McIntosh. 2018. Noise and heterogeneity in historical build data: An empirical study of Travis CI. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 87–97.

[18] Lina Gong, Shujuan Jiang, Rongcun Wang, and Li Jiang. 2019. Empirical evaluation of the impact of class overlap on software defect prediction. In *Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 698–709.

[19] Lina Gong, Haoxiang Zhang, Jingxuan Zhang, Mingqiang Wei, and Zhiqiu Huang. 2022. A comprehensive investigation of the impact of class overlap on software defect prediction. *IEEE Transactions on Software Engineering* 49, 4 (2022), 2440–2458.

[20] Donghai Guan, Weiwei Yuan, Young-Koo Lee, and Sungyoung Lee. 2011. Identifying mislabeled training data with the aid of unlabeled data. *Applied Intelligence* 35, 3 (2011), 345–358.

[21] Shivani Gupta and Atul Gupta. 2019. Dealing with noise problem in machine learning data-sets: A systematic review. *Procedia Computer Science* 161 (2019), 466–474.

[22] Emitza Guzman, David Azócar, and Yang Li. 2014. Sentiment analysis of commit comments in GitHub: An empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, 352–355.

[23] Aram Hovsepyan, Riccardo Scandariato, Wouter Joosen, and James Walden. 2012. Software vulnerability prediction using text analysis techniques. In *Proceedings of the 4th International Workshop on Security Measurements and Metrics*, 7–10.

[24] Faria Huq, Masum Hasan, Mahim Anzum Pantho Haque, Sazan Mahbub, Anindya Iqbal, and Toufique Ahmed. 2021. *Review4Repair: Code Review Aided Automatic Program Repairing*. DOI: https://doi.org/10.5281/zenodo.4445747

[25] Robbert Jongeling, Subhajit Datta, and Alexander Serebrenik. 2015. Choosing your weapons: On sentiment analysis tools for software engineering research. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 531–535.

[26] Taghi M. Khoshgoftaar and Pierre Rebours. 2007. Improving software quality prediction by noise filtering techniques. *Journal of Computer Science and Technology* 22, 3 (2007), 387–396.

[27] Taghi M. Khoshgoftaar, Naeem Seliya, and Kehan Gao. 2005. Detecting noisy instances with the rule-based classification model. *Intelligent Data Analysis* 9, 4 (2005), 347–364.

[28] Sunghun Kim, Hongyu Zhang, Rongxin Wu, and Liang Gong. 2011. Dealing with noise in defect prediction. In *Proceedings of the 2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 481–490.

[29] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. arXiv:1412.6980. Retrieved from https://arxiv.org/abs/1412.6980

[30] Miikka Kuutila, Mika Mäntylä, Umar Farooq, and Maelick Claes. 2020. Time pressure in software engineering: A systematic review. *Information and Software Technology* 121 (2020), 106257.

[31] Gernot Liebchen, Bheki Twala, Martin Shepperd, Michelle Cartwright, and Mark Stephens. 2007. Filtering, robust filtering, polishing: Techniques for addressing quality in software data. In *Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement (ESEM '07)*. IEEE, 99–106.

[32] Gernot Armin Liebchen. 2010. *Data Cleaning Techniques for Software Engineering Data Sets*. Ph.D. Dissertation. School of Information Systems, Computing and Mathematics, Brunel University.

[33] Jediael Mendoza, Jason Mycroft, Lyam Milbury, Nafiseh Kahani, and Jason Jaskolka. 2022. On the effectiveness of data balancing techniques in the context of ML-based test case prioritization. In *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*, 72–81.

[34] Fabrice Muhlenbach, Stéphane Lallich, and Djamel A. Zighed. 2004. Identifying and handling mislabelled instances. *Journal of Intelligent Information Systems* 22, 1 (01 Jan. 2004), 89–109. DOI: https://doi.org/10.1023/A:1025832930864

[35] Curtis Northcutt, Lu Jiang, and Isaac Chuang. 2021. Confident learning: Estimating uncertainty in dataset labels. *Journal of Artificial Intelligence Research* 70 (2021), 1373–1411.

[36] Miroslaw Ochodek, Miroslaw Staron, Dominik Bargowski, Wilhelm Meding, and Regina Hebig. 2017. Using machine learning to design a flexible LOC counter. In *Proceedings of the 2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*. IEEE, 14–20.

[37] Cong Pan, Minyan Lu, and Biao Xu. 2021. An empirical study on software defect prediction using CodeBERT model. *Applied Sciences* 11, 11 (2021), 4793.

[38] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[39] Nakul Pritam, Manju Khari, Le Hoang Son, Raghvendra Kumar, Sudan Jha, Ishaani Priyadarshini, Mohamed Abdel-Basset, and Hoang Viet Long, et al. 2019. Assessment of code smell for predicting class change proneness using machine learning. *IEEE Access* 7 (2019), 37414–37425.

[40] Philipp Probst, Marvin N. Wright, and Anne-Laure Boulesteix. 2019. Hyperparameters and tuning strategies for random Forest. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 9, 3 (2019), e1301.

[41] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research* 21, 1 (2020), 5485–5551.

[42] Mohammad Masudur Rahman, Chanchal K. Roy, and Raula G. Kula. 2017. Predicting usefulness of code review comments using textual features and developer experience. In *Proceedings of the 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 215–226.

[43] Pierre Rebours and Taghi M. Khoshgoftaar. 2006. Quality problem in software measurement data. In *Advances in Computers*. Marvin V. Zelkowitz (Ed.), Vol. 66. Elsevier, 43–77.

[44] Athanasios-Ilias Rousinopoulos, Gregorio Robles, and Jesús M. González-Barahona. 2014. Sentiment analysis of free/open source developers: Preliminary findings from a case study. *Revista Eletrônica de Sistemas de Informação* 13, 2, Article 6 (2014), 1–21.

[45] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. 2018. Modern code review: A case study at Google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, 181–190.

[46] Islem Saidani, Ali Ouni, and Mohamed Wiem Mkaouer. 2021. Detecting continuous integration skip commits using multi-objective evolutionary search. *IEEE Transactions on Software Engineering* 48, 12 (2021), 4873–4891.

[47] Chris Seiffert, Taghi M. Khoshgoftaar, Jason Van Hulse, and Andres Folleco. 2014. An empirical study of the classification performance of learners on imbalanced and noisy software quality data. *Information Sciences* 259 (2014), 571–595.

[48] Yogesh Singh, Pradeep Kumar Bhatia, Arvinder Kaur, and Omprakash Sangwan. 2009. Application of neural networks in software engineering: A review. In *Proceedings of the 3rd International Conference on Information Systems, Technology and Management (ICISTM '09)*. Springer, 128–137.

[49] Borut Sluban and Nada Lavrač. 2015. Relating ensemble diversity and performance: A study in class noise detection. *Neurocomputing* 160 (2015), 120–131.

[50] Mulia Kevin Suryadi, Rudy Herteno, Setyo Wahyu Saputro, Mohammad Reza Faisal, and Radityo Adi Nugroho. 2024. Comparative study of various hyperparameter tuning on random Forest classification with SMOTE and feature selection using genetic algorithm in software defect prediction. *Journal of Electronics, Electromedical Engineering, and Medical Informatics* 6, 2 (2024), 137–147.

[51] Wei Tang and Taghi M. Khoshgoftaar. 2004. Noise identification with the k-means algorithm. In *Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence*. IEEE, 373–378.

[52] Choh-Man Teng. 1999. Correcting noisy data. In *Proceedings of the 16th International Conference on Machine Learning (ICML '99)*. Citeseer, 239–248.

[53] Mike Thelwall, Kevan Buckley, Georgios Paltoglou, Di Cai, and Arvid Kappas. 2010. Sentiment strength detection in short informal text. *Journal of the American Society for Information Science and Technology* 61, 12 (2010), 2544–2558.

[54] Jason Van Hulse, Taghi M. Khoshgoftaar, Chris Seiffert, and Lili Zhao. 2006. Noise correction using Bayesian multiple imputation. In *Proceedings of the 2006 IEEE International Conference on Information Reuse and Integration*. IEEE, 478–483.

[55] Leonardo Villalobos-Arias and Christian Quesada-López. 2021. Comparative study of random search hyper-parameter tuning for software effort estimation. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, 21–29.

[56] D. Randall Wilson and Tony R. Martinez. 2000. Reduction techniques for instance-based learning algorithms. *Machine Learning* 38, 3 (2000), 257–286.

[57] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering*. Springer Science & Business Media.

[58] Jing Xia and Yanhui Li. 2017. Could we predict the result of a continuous integration build? An empirical study. In *Proceedings of the 2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 311–315.

[59] Jing Xia, Yanhui Li, and Chuanqi Wang. 2017. An empirical study on the cross-project predictability of continuous integration outcomes. In *Proceedings of the 2017 14th Web Information Systems and Applications Conference (WISA)*. IEEE, 234–239.

[60] Shi Zhong, Taghi M. Khoshgoftaar, and Naeem Seliya. 2004. Analyzing software measurement data with clustering techniques. *IEEE Intelligent Systems* 19, 2 (2004), 20–27.

[61] Xin Zhou, DongGyun Han, and David Lo. 2021. Assessing generalizability of CodeBERT. In *Proceedings of the 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 425–436.

[62] Xingquan Zhu and Xindong Wu. 2004. Class noise vs. attribute noise: A quantitative study. *Artificial Intelligence Review* 22, 3 (2004), 177–210.

Appendix A. Detailed Evaluation Metrics for All Experiments

Table A1. The Evaluation Results for Comparing the Effectiveness of RF, Extended Gradient Boosting, and NN

Index	noise_alg	Classifier	Precision	Recall	F1	MCC
0	CF	RF	0.85	0.86	0.85	0.23
1	CF	nn	0.85	0.86	0.85	0.22
2	CF	xgb	0.83	0.91	0.84	0.17
3	DB	RF	0.83	0.83	0.8	0.12
4	DB	nn	0.83	0.64	0.69	0.15
5	DB	xgb	0.85	0.64	0.68	0.15
6	MF	RF	0.85	0.88	0.86	0.26
7	MF	nn	0.85	0.88	0.86	0.22
8	MF	xgb	0.83	0.9	0.84	0.15
9	none	RF	0.86	0.9	0.87	0.23
10	none	nn	0.86	0.88	0.87	0.2
11	none	xgb	0.85	0.92	0.87	0.17

Table A2. Descriptive Statistics of the Dependent Variables for an RF Model before Applying Any Treatment Levels

Project	Precision		Recall		F1		MCC	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD
AcDisplay	0.47	0.02	0.55	0.03	0.51	0.02	0.06	0.04
DDT	0.83	0.02	0.55	0.15	0.66	0.11	0.05	0.08
HearthSim	1.0	0.0	0.62	0.14	0.76	0.1	0.04	0.03
HikariCP	0.91	0.03	0.66	0.2	0.75	0.15	0.17	0.17
Hydra	0.94	0.01	0.7	0.1	0.8	0.07	0.09	0.08
Hystrix	0.63	0.07	0.67	0.13	0.65	0.09	0.15	0.17
Jest	0.89	0.01	0.62	0.09	0.72	0.06	0.0	0.06
LittleProxy	0.9	0.01	0.69	0.11	0.78	0.08	0.06	0.05
MozStumbler	1.0	0.0	0.97	0.02	0.98	0.01	0.09	0.13
OpenRefine	0.96	0.03	0.86	0.06	0.91	0.04	0.33	0.22
ProjectRed	0.88	0.08	0.85	0.17	0.86	0.12	0.67	0.23
RoaringBitmap	0.99	0.0	0.74	0.08	0.85	0.05	0.15	0.05
Singularity	0.79	0.09	0.59	0.06	0.68	0.06	0.13	0.19
Dspace	0.97	0.0	0.99	0.01	0.98	0.01	0.08	0.16
auto	0.99	0.0	1.00	0.00	0.99	0.00	0.16	0.17
airlift	0.68	0.24	0.87	0.05	0.74	0.15	0.51	0.32
analytics-android	0.95	0.02	0.81	0.09	0.87	0.06	0.17	0.17
android	0.94	0.07	0.8	0.06	0.86	0.05	0.62	0.19
android-maven-plugin	0.96	0.01	0.82	0.16	0.88	0.11	0.17	0.16
assertj-android	0.92	0.05	0.57	0.18	0.69	0.14	0.38	0.17
basex	0.97	0.0	0.79	0.04	0.87	0.02	0.01	0.01

(Continued)

Table A2. Continued

Project	Precision		Recall		F1		MCC	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD
blueflood	0.93	0.03	0.67	0.07	0.77	0.05	0.14	0.13
blueprints	0.71	0.1	0.73	0.11	0.71	0.04	0.16	0.19
bnd	0.93	0.02	0.64	0.08	0.76	0.05	0.14	0.07
brightspot-cms	0.87	0.08	0.85	0.06	0.85	0.04	0.51	0.2
cas-addons	1.0	0.0	0.91	0.02	0.95	0.01	0.13	0.13
cassandra-reaper	0.88	0.04	0.68	0.12	0.76	0.1	0.07	0.14
ccw	0.76	0.02	0.72	0.05	0.74	0.03	0.08	0.08
checkstyle	1.0	0.0	0.87	0.06	0.93	0.03	0.04	0.05
cloudify	0.82	0.02	0.56	0.11	0.66	0.07	0.04	0.07
core	1.0	0.0	0.58	0.05	0.73	0.04	0.02	0.03
dagger	0.97	0.01	0.77	0.05	0.86	0.03	0.05	0.07
dropwizard	0.99	0.01	0.75	0.06	0.85	0.04	0.09	0.08
dynjs	0.98	0.01	0.73	0.16	0.83	0.1	0.17	0.27
error-prone	1.0	0.0	0.99	0.01	0.99	0.0	0.07	0.05
frontend-maven-plugin	0.98	0.01	0.8	0.06	0.88	0.04	0.19	0.12
go-lang-idea-plugin	0.97	0.01	0.76	0.14	0.85	0.09	0.13	0.08
goclipse	0.99	0.0	0.8	0.06	0.89	0.04	0.03	0.02
gpslogger	0.96	0.01	0.77	0.08	0.85	0.05	0.18	0.11
hivemall	0.99	0.0	0.88	0.06	0.93	0.03	0.06	0.06
htm.java	1.0	0.0	0.88	0.09	0.93	0.05	0.28	0.3
idea-gitignore	0.85	0.02	0.64	0.11	0.73	0.09	0.04	0.08
jInstagram	1.0	0.0	0.76	0.14	0.85	0.1	0.17	0.09
jPOS	1.0	0.0	0.79	0.1	0.88	0.06	0.0	0.04
jade4j	0.98	0.01	0.65	0.23	0.76	0.18	0.02	0.07
javaslang	1.0	0.0	0.91	0.13	0.95	0.08	0.26	0.22
jcabi-aspects	0.89	0.04	0.65	0.17	0.74	0.12	0.19	0.14
jcabi-github	0.76	0.14	0.77	0.08	0.76	0.08	0.23	0.34
jcabi-http	0.86	0.03	0.62	0.12	0.72	0.09	−0.01	0.1
jedis	0.99	0.0	0.85	0.13	0.91	0.08	0.19	0.12
jmeter-plugins	0.92	0.06	0.96	0.02	0.94	0.04	0.46	0.43
jmonkeyengine	0.99	0.0	0.8	0.02	0.89	0.01	0.1	0.06
jmxtrans	0.98	0.0	0.76	0.07	0.86	0.04	0.07	0.07
joda-time	1.0	0.0	0.88	0.05	0.94	0.03	0.02	0.04
jodd	0.99	0.0	0.7	0.22	0.8	0.16	0.11	0.13
jphp	0.9	0.04	0.64	0.14	0.75	0.09	0.15	0.22
jsonld-java	1.0	0.0	0.94	0.04	0.97	0.02	0.13	0.09
jsprit	1.0	0.0	0.89	0.06	0.94	0.03	0.16	0.09
keywhiz	1.0	0.0	0.99	0.01	1.0	0.0	−0.0	0.0
lenskit	0.99	0.0	0.81	0.12	0.88	0.09	0.01	0.04
less4j	0.92	0.03	0.85	0.03	0.89	0.02	0.17	0.2
logback	0.98	0.01	0.73	0.08	0.83	0.06	0.08	0.06
lorsource	0.98	0.0	0.72	0.09	0.83	0.06	0.05	0.05

(Continued)

Table A2. Continued

Project	Precision		Recall		F1		MCC	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD
maven-git-commit-id-plugin	0.92	0.02	0.64	0.13	0.75	0.1	0.15	0.14
metrics	0.91	0.05	0.8	0.08	0.85	0.04	0.37	0.18
mockito	0.97	0.01	0.86	0.11	0.91	0.06	0.19	0.1
mybatis-3	1.0	0.0	0.75	0.1	0.85	0.07	0.07	0.04
nokogiri	0.71	0.11	0.76	0.12	0.73	0.09	0.28	0.24
nutz	0.66	0.03	0.61	0.06	0.63	0.04	0.01	0.08
okhttp	0.76	0.03	0.69	0.05	0.72	0.03	−0.01	0.09
onebusaway-android	1.0	0.0	0.93	0.06	0.96	0.03	0.16	0.16
openwayback	1.0	0.0	0.97	0.02	0.99	0.01	0.14	0.13
owner	0.99	0.0	0.81	0.06	0.89	0.04	0.04	0.08
p6spy	0.84	0.16	0.82	0.08	0.81	0.06	0.53	0.28
parceler	0.99	0.0	0.68	0.1	0.8	0.08	0.07	0.05
pdfsam	0.8	0.14	0.56	0.2	0.65	0.19	0.02	0.22
play-authenticate	0.97	0.01	0.91	0.08	0.94	0.05	0.36	0.22
psi-probe	1.0	0.0	0.92	0.08	0.96	0.04	0.14	0.21
pushy	1.0	0.0	0.99	0.01	0.99	0.0	0.37	0.33
querydsl	0.98	0.01	0.75	0.12	0.85	0.08	0.05	0.08
quickml	0.98	0.01	0.71	0.06	0.82	0.04	0.12	0.1
qulice	0.98	0.0	0.8	0.05	0.88	0.03	−0.01	0.05
restlet-framework-java	0.57	0.06	0.54	0.16	0.55	0.11	0.06	0.15
retrofit	0.99	0.0	0.89	0.04	0.94	0.02	0.15	0.1
rewrite	0.76	0.06	0.57	0.29	0.63	0.2	0.19	0.27
rexster	0.99	0.01	0.79	0.15	0.88	0.09	0.12	0.17
robospice	0.46	0.06	0.48	0.11	0.46	0.07	0.04	0.11
rultor	0.72	0.02	0.62	0.12	0.66	0.07	0.07	0.04
rxjava-jdbc	1.0	0.0	0.84	0.04	0.91	0.02	0.03	0.04
selendroid	0.73	0.03	0.68	0.12	0.7	0.08	0.12	0.09
seyren	0.99	0.01	0.73	0.07	0.84	0.05	0.14	0.09
sms-backup-plus	0.99	0.0	0.77	0.09	0.86	0.06	0.0	0.03
spark	0.99	0.0	0.83	0.04	0.9	0.02	0.08	0.07
spring-cloud-config	0.95	0.02	0.62	0.14	0.74	0.1	0.02	0.08
springside4	0.73	0.07	0.56	0.15	0.63	0.12	0.17	0.15
storio	0.97	0.02	0.65	0.07	0.78	0.05	0.17	0.11
storm	0.52	0.13	0.57	0.17	0.53	0.1	0.17	0.19
structr	0.57	0.12	0.43	0.11	0.48	0.1	0.18	0.17
stubby4j	0.75	0.07	0.6	0.09	0.66	0.06	0.2	0.12
thredds	0.74	0.03	0.64	0.08	0.69	0.06	−0.0	0.11
traccar	1.0	0.0	0.84	0.02	0.91	0.01	0.05	0.03
truth	0.97	0.02	0.74	0.06	0.84	0.04	0.26	0.14
twilio-java	0.85	0.03	0.67	0.08	0.75	0.05	0.14	0.11
u2020	0.98	0.01	0.75	0.1	0.85	0.07	0.13	0.13
unirest-java	0.93	0.01	0.59	0.07	0.72	0.05	−0.0	0.03

(Continued)

Table A2. Continued

Project	Precision		Recall		F1		MCC	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD
waffle	1.0	0.0	0.86	0.06	0.92	0.03	0.1	0.08
webcam-capture	1.0	0.0	0.89	0.04	0.94	0.02	0.07	0.04
wire	1.0	0.0	0.98	0.02	0.99	0.01	0.03	0.05
xtreemfs	0.98	0.02	0.83	0.12	0.9	0.08	0.31	0.27
yobi	1.0	0.0	0.9	0.05	0.94	0.03	0.06	0.03

Table A3. Descriptive Statistics of the Dependent Variables for an RF Model after Applying DB

Project	Precision		Recall		F1		MCC	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD
AcDisplay	0.45	0.02	0.86	0.04	0.59	0.02	0.01	0.08
DDT	0.83	0.03	0.54	0.12	0.65	0.09	0.05	0.09
DSpace	0.98	0.0	0.9	0.09	0.93	0.05	0.05	0.05
HearthSim	0.99	0.0	0.92	0.06	0.96	0.04	0.0	0.02
HikariCP	0.87	0.01	0.85	0.07	0.86	0.04	−0.05	0.08
Hydra	0.93	0.01	0.92	0.02	0.92	0.01	0.06	0.07
Hystrix	0.57	0.03	0.61	0.07	0.59	0.05	−0.01	0.09
Jest	0.89	0.01	0.82	0.08	0.85	0.05	0.0	0.05
LittleProxy	0.89	0.01	0.68	0.06	0.77	0.04	0.04	0.06
MozStumbler	1.0	0.0	0.98	0.01	0.99	0.01	0.08	0.17
OpenRefine	0.91	0.01	0.83	0.06	0.87	0.04	−0.03	0.08
ProjectRed	0.91	0.1	0.81	0.16	0.85	0.11	0.67	0.22
RoaringBitmap	0.98	0.0	0.99	0.01	0.99	0.0	0.01	0.04
Singularity	0.79	0.06	0.66	0.08	0.71	0.05	0.14	0.14
DSpace	0.98	0.0	0.90	0.09	0.93	0.05	0.05	0.05
auto	0.99	0.0	0.88	0.03	0.93	0.02	0.06	0.06
airlift	0.63	0.23	0.14	0.12	0.23	0.17	0.16	0.18
analytics-android	0.94	0.01	0.87	0.08	0.9	0.05	0.13	0.15
android-maven-plugin	0.95	0.01	0.77	0.06	0.85	0.04	0.06	0.05
android	0.95	0.06	0.83	0.05	0.88	0.04	0.67	0.15
assertj-android	0.91	0.04	0.66	0.19	0.75	0.13	0.41	0.16
auto	0.99	0.0	0.88	0.03	0.93	0.02	0.06	0.06
basex	0.97	0.0	0.86	0.05	0.91	0.03	0.04	0.05
blueflood	0.89	0.01	0.81	0.06	0.85	0.03	0.02	0.06
blueprints	0.71	0.11	0.68	0.08	0.69	0.06	0.16	0.22
bnd	0.93	0.02	0.75	0.07	0.83	0.04	0.18	0.1
brightspot-cms	0.84	0.1	0.89	0.04	0.86	0.05	0.43	0.31
cas-addons	1.0	0.0	0.94	0.02	0.97	0.01	0.2	0.11
cassandra-reaper	0.89	0.03	0.71	0.13	0.79	0.1	0.09	0.13
ccw	0.76	0.03	0.81	0.07	0.78	0.04	0.09	0.13

(Continued)

Table A3. Continued

Project	Precision		Recall		F1		MCC	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD
checkstyle	1.0	0.0	0.99	0.01	1.0	0.0	−0.0	0.0
cloudify	0.84	0.02	0.57	0.07	0.68	0.05	0.04	0.05
core	1.0	0.0	0.68	0.08	0.81	0.05	0.02	0.04
dagger	0.97	0.01	0.83	0.05	0.9	0.03	0.07	0.09
dropwizard	0.98	0.0	0.72	0.05	0.83	0.04	−0.01	0.02
dynjs	0.98	0.01	0.85	0.09	0.91	0.05	0.11	0.12
error-prone	1.0	0.0	0.99	0.0	1.0	0.0	0.09	0.08
frontend-maven-plugin	0.96	0.01	0.92	0.06	0.94	0.03	0.03	0.12
go-lang-idea-plugin	0.96	0.02	0.72	0.14	0.82	0.1	0.03	0.18
goclipse	0.99	0.0	0.89	0.03	0.94	0.01	0.03	0.02
gpslogger	0.96	0.01	0.85	0.05	0.9	0.03	0.17	0.14
hivemall	0.99	0.0	0.89	0.06	0.94	0.04	−0.0	0.04
htm.java	0.99	0.0	0.96	0.04	0.98	0.02	0.04	0.06
idea-gitignore	0.83	0.02	0.74	0.19	0.77	0.13	−0.01	0.09
jInstagram	1.0	0.0	0.76	0.15	0.85	0.1	0.18	0.09
jPOS	1.0	0.0	0.88	0.09	0.93	0.05	0.0	0.03
jade4j	0.98	0.01	0.73	0.26	0.81	0.19	−0.02	0.08
javaslang	1.0	0.0	0.93	0.09	0.96	0.05	0.16	0.16
jcabi-aspects	0.83	0.03	0.74	0.13	0.78	0.08	−0.02	0.13
jcabi-github	0.65	0.05	0.56	0.1	0.6	0.08	−0.02	0.12
jcabi-http	0.85	0.04	0.66	0.11	0.74	0.08	−0.05	0.13
jedis	0.98	0.0	0.9	0.11	0.94	0.06	0.17	0.12
jmeter-plugins	0.86	0.09	0.61	0.29	0.69	0.22	0.14	0.4
jmonkeyengine	0.99	0.0	0.97	0.01	0.98	0.01	−0.0	0.02
jmxtrans	0.98	0.0	0.97	0.02	0.97	0.01	−0.0	0.03
joda-time	1.0	0.0	0.94	0.03	0.97	0.01	−0.01	0.01
jodd	0.99	0.0	0.78	0.18	0.86	0.11	0.07	0.07
jphp	0.91	0.03	0.77	0.16	0.82	0.11	0.21	0.24
jsonld-java	1.0	0.0	0.97	0.03	0.98	0.02	0.12	0.11
jsprit	1.0	0.0	0.92	0.06	0.95	0.03	0.01	0.04
keywhiz	1.0	0.0	1.0	0.0	1.0	0.0	−0.0	0.0
lenskit	0.99	0.0	0.87	0.08	0.92	0.05	0.02	0.04
less4j	0.89	0.01	0.73	0.06	0.8	0.04	−0.03	0.07
logback	0.98	0.0	0.83	0.06	0.9	0.04	0.08	0.06
lorsource	0.98	0.0	0.79	0.07	0.87	0.04	0.03	0.04
maven-git-commit-id-plugin	0.92	0.03	0.61	0.16	0.72	0.12	0.15	0.16
metrics	0.9	0.05	0.85	0.07	0.87	0.05	0.4	0.2
mockito	0.97	0.01	0.9	0.08	0.93	0.04	0.13	0.14
mybatis-3	0.99	0.0	0.68	0.04	0.8	0.03	0.01	0.06
nokogiri	0.71	0.16	0.47	0.32	0.53	0.26	0.23	0.29
nutz	0.65	0.02	0.64	0.06	0.64	0.04	−0.0	0.05

(Continued)

Table A3. Continued

Project	Precision		Recall		F1		MCC	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD
okhttp	0.76	0.02	0.74	0.06	0.75	0.03	−0.0	0.08
onebusaway-android	1.0	0.0	0.97	0.03	0.98	0.02	0.08	0.09
openwayback	1.0	0.0	0.99	0.01	1.0	0.0	0.24	0.27
owner	0.99	0.0	0.97	0.04	0.98	0.02	−0.01	0.02
p6spy	0.83	0.16	0.8	0.08	0.8	0.07	0.51	0.28
parceler	0.99	0.0	0.71	0.09	0.82	0.07	0.06	0.05
pdfsam	0.82	0.04	0.82	0.18	0.81	0.1	0.02	0.23
play-authenticate	0.95	0.02	0.88	0.11	0.91	0.06	0.12	0.17
psi-probe	1.0	0.0	0.96	0.03	0.98	0.02	0.21	0.19
pushy	1.0	0.0	0.99	0.01	0.99	0.0	0.1	0.24
querydsl	0.97	0.01	0.75	0.17	0.84	0.12	−0.02	0.05
quickml	0.98	0.01	0.76	0.05	0.85	0.03	0.12	0.12
qulice	0.98	0.0	0.78	0.05	0.87	0.03	−0.03	0.04
restlet-framework-java	0.52	0.31	0.24	0.39	0.26	0.37	0.09	0.45
retrofit	0.99	0.0	0.93	0.04	0.96	0.02	0.13	0.09
rewrite	0.78	0.08	0.48	0.35	0.54	0.28	0.21	0.3
rexster	0.99	0.01	0.86	0.1	0.92	0.06	0.14	0.17
robospice	0.47	0.14	0.26	0.07	0.32	0.07	0.02	0.14
rultor	0.69	0.06	0.5	0.08	0.58	0.06	−0.0	0.14
rxjava-jdbc	1.0	0.0	0.87	0.04	0.93	0.02	0.04	0.04
selendroid	0.75	0.07	0.54	0.12	0.62	0.09	0.09	0.14
seyren	0.98	0.0	0.81	0.05	0.89	0.03	0.02	0.04
sms-backup-plus	0.99	0.0	0.95	0.03	0.97	0.01	0.01	0.05
spark	0.99	0.0	0.82	0.08	0.89	0.05	0.05	0.05
spring-cloud-config	0.95	0.01	0.72	0.09	0.82	0.06	0.04	0.09
springside4	0.72	0.07	0.66	0.14	0.68	0.1	0.16	0.19
storio	0.98	0.02	0.66	0.08	0.79	0.05	0.19	0.11
storm	0.4	0.08	0.43	0.14	0.41	0.1	−0.02	0.15
structr	0.42	0.12	0.21	0.14	0.26	0.12	−0.0	0.1
stubby4j	0.75	0.1	0.57	0.09	0.64	0.08	0.2	0.19
thredds	0.74	0.04	0.63	0.09	0.68	0.06	−0.0	0.12
traccar	1.0	0.0	0.89	0.03	0.94	0.02	0.04	0.03
truth	0.93	0.0	0.98	0.02	0.95	0.01	−0.03	0.02
twilio-java	0.85	0.05	0.59	0.1	0.69	0.08	0.14	0.16
u2020	0.98	0.01	0.9	0.08	0.93	0.05	0.09	0.11
unirest-java	0.94	0.01	0.88	0.08	0.9	0.05	0.03	0.14
waffle	0.99	0.0	0.87	0.05	0.93	0.03	0.05	0.03
webcam-capture	0.99	0.0	0.89	0.08	0.94	0.05	0.07	0.05
wire	1.0	0.0	0.99	0.0	1.0	0.0	0.04	0.09
xtreemfs	0.96	0.0	0.97	0.02	0.96	0.01	0.0	0.05
yobi	1.0	0.0	0.91	0.05	0.95	0.03	0.03	0.05

Table A4. Descriptive Statistics of the Dependent Variables for an RF Model after Applying CleanLab

Project	Precision		Recall		F1		MCC	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD
AcDisplay	0.52	0.05	0.38	0.04	0.44	0.03	0.10	0.06
DDT	0.82	0.01	0.94	0.03	0.87	0.02	0.06	0.08
DSpace	0.97	0	0.99	0.01	0.98	0.01	0.10	0.14
HearthSim	0.99	0	0.99	0.01	0.99	0	0.01	0.02
HikariCP	0.88	0.01	0.94	0.07	0.91	0.03	0.06	0.10
Hydra	0.93	0.01	0.98	0.02	0.96	0.01	0.26	0.17
Hystrix	0.63	0.08	0.72	0.15	0.67	0.10	0.17	0.21
Jest	0.90	0.01	0.95	0.05	0.92	0.02	0.10	0.11
LittleProxy	0.89	0.01	0.95	0.06	0.92	0.03	0.11	0.08
MozStumbler	1	0	1	0	1	0	0.17	0.33
OpenRefine	0.96	0.03	0.98	0.02	0.97	0.01	0.55	0.24
ProjectRed	0.89	0.10	0.85	0.17	0.86	0.13	0.67	0.27
RoaringBitmap	0.98	0	0.99	0.01	0.99	0	0.18	0.10
Singularity	0.77	0.05	0.89	0.04	0.82	0.04	0.15	0.22
auto	0.99	0	1	0	0.99	0	0.09	0.11
airlift	0.69	0.23	0.84	0.04	0.74	0.14	0.51	0.31
analytics-android	0.94	0.01	0.97	0.04	0.95	0.02	0.14	0.22
android-maven-plugin	0.95	0.01	0.94	0.12	0.94	0.07	0.18	0.16
android	0.88	0.05	0.95	0.03	0.91	0.03	0.68	0.15
assertj-android	0.83	0.02	0.88	0.07	0.85	0.03	0.33	0.09
basex	0.97	0	0.99	0.01	0.98	0	0.04	0.05
blueflood	0.91	0.01	0.97	0.02	0.94	0.01	0.18	0.14
blueprints	0.68	0.06	0.80	0.10	0.73	0.04	0.12	0.16
bnd	0.91	0.01	0.98	0.01	0.94	0.01	0.24	0.14
brightspot-cms	0.86	0.07	0.89	0.04	0.87	0.04	0.53	0.22
cas-addons	0.99	0	1	0	1	0	0.44	0.39
cassandra-reaper	0.88	0.01	0.95	0.03	0.91	0.01	0.10	0.11
ccw	0.75	0.01	0.88	0.04	0.81	0.02	0.08	0.08
checkstyle	1	0	1	0	1	0	0.15	0.19
cloudify	0.83	0	0.92	0.05	0.87	0.02	0.02	0.03
core	1	0	1	0	1	0	0.06	0.06
dagger	0.97	0	1	0	0.98	0	0.09	0.13
dropwizard	0.98	0	0.99	0	0.99	0	0.13	0.13
dynjs	0.97	0	0.99	0.01	0.98	0	0.20	0.17
error-prone	1	0	1	0	1	0	0	0
frontend-maven-plugin	0.97	0.01	1	0	0.98	0	0.32	0.28
go-lang-idea-plugin	0.97	0.01	0.99	0.01	0.98	0.01	0.30	0.25
goclipse	0.99	0	1	0	0.99	0	0.05	0.04
gpslogger	0.95	0.01	0.98	0.03	0.96	0.01	0.23	0.21
hivemall	0.99	0	1	0.01	0.99	0	0.08	0.14
htm,java	0.99	0	1	0	1	0	0.39	0.32

(Continued)

Table A4. Continued

Project	Precision		Recall		F1		MCC	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD
idea-gitignore	0.84	0.01	0.93	0.06	0.88	0.03	0.03	0.05
jInstagram	0.99	0	1	0.01	0.99	0	0.36	0.19
jPOS	1	0	1	0	1	0	0.17	0.14
jade4j	0.98	0	0.93	0.06	0.96	0.04	0	0.04
javaslang	0.99	0	1	0.01	0.99	0	0.11	0.16
jcabi-aspects	0.86	0.01	0.95	0.04	0.90	0.02	0.19	0.15
jcabi-github	0.76	0.12	0.86	0.09	0.80	0.08	0.29	0.33
jcabi-http	0.88	0.03	0.93	0.04	0.90	0.03	0.13	0.25
jedis	0.98	0	0.98	0.03	0.98	0.02	0.04	0.06
jmeter-plugins	0.93	0.05	0.97	0.02	0.95	0.03	0.51	0.38
jmonkeyengine	0.99	0	1	0.01	0.99	0	0.20	0.24
jmxtrans	0.98	0	0.99	0.01	0.99	0	0.09	0.11
joda-time	1	0	1	0.01	1	0	0	0
jodd	0.99	0	1	0	0.99	0	0.09	0.15
jphp	0.89	0.02	0.96	0.04	0.92	0.02	0.20	0.22
jsonld-java	1	0	1	0	1	0	0.10	0.15
jsprit	1	0	1	0.01	1	0	0.46	0.37
keywhiz	1	0	1	0	1	0	0	0
lenskit	0.99	0	0.99	0	0.99	0	0.09	0.12
less4j	0.93	0.05	0.98	0.02	0.96	0.03	0.37	0.47
logback	0.97	0	0.99	0.02	0.98	0.01	0.17	0.09
lorsource	0.98	0	0.99	0	0.99	0	0.07	0.10
maven-git-commit-id-plugin	0.90	0.01	0.95	0.04	0.92	0.02	0.12	0.13
metrics	0.88	0.05	0.92	0.06	0.90	0.03	0.36	0.27
mockito	0.96	0	0.98	0.02	0.97	0.01	0.18	0.17
mybatis-3	1	0	1	0	1	0	0.29	0.25
nokogiri	0.71	0.11	0.81	0.12	0.75	0.09	0.31	0.29
nutz	0.65	0.01	0.83	0.04	0.73	0.02	−0.02	0.06
okhttp	0.76	0.02	0.89	0.05	0.82	0.03	0.03	0.14
onebusaway-android	1	0	1	0	1	0	0.17	0.30
openwayback	1	0	1	0	1	0	0.25	0.30
owner	0.99	0	0.99	0.01	0.99	0	0.06	0.08
p6spy	0.83	0.16	0.85	0.06	0.83	0.07	0.54	0.29
parceler	0.98	0	1	0.01	0.99	0	0.04	0.07
pdfsam	0.84	0.02	0.89	0.06	0.86	0.04	0.13	0.16
play-authenticate	0.95	0.01	0.96	0.06	0.96	0.04	0.26	0.25
psi-probe	1	0	1	0	1	0	0.59	0.38
pushy	1	0	1	0	1	0	0.20	0.42
querydsl	0.98	0	1	0	0.99	0	0.13	0.21
quickml	0.97	0.01	0.99	0.01	0.98	0	0.19	0.22
qulice	0.98	0	0.99	0.01	0.98	0	−0.01	0.01
restlet-framework-java	0.59	0.07	0.71	0.18	0.64	0.11	0.13	0.22

(Continued)

Table A4. Continued

Project	Precision		Recall		F1		MCC	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD
retrofit	0.98	0	1	0.01	0.99	0	0.23	0.17
rewrite	0.76	0.03	0.83	0.13	0.79	0.06	0.19	0.18
rexster	0.99	0	1	0	0.99	0	0.23	0.31
robospice	0.49	0.09	0.41	0.11	0.44	0.07	0.07	0.13
rultor	0.71	0.03	0.88	0.07	0.78	0.03	0.09	0.14
rxjava-jdbc	1	0	0.99	0.01	1	0.01	0.06	0.13
selendroid	0.72	0.02	0.86	0.06	0.78	0.03	0.12	0.10
seyren	0.98	0	0.98	0.01	0.98	0.01	0.22	0.17
sms-backup-plus	0.99	0	0.99	0.01	0.99	0	−0.01	0.02
spark	0.99	0	0.99	0	0.99	0	0.03	0.08
spring-cloud-config	0.95	0	0.98	0.01	0.96	0.01	0.03	0.04
springside4	0.70	0.02	0.83	0.12	0.75	0.07	0.16	0.12
storio	0.95	0.01	0.98	0.02	0.97	0.01	0.33	0.19
storm	0.54	0.17	0.43	0.20	0.46	0.16	0.17	0.25
structr	0.58	0.11	0.41	0.12	0.47	0.11	0.18	0.16
stubby4j	0.71	0.04	0.84	0.06	0.77	0.03	0.23	0.11
thredds	0.75	0.03	0.86	0.08	0.80	0.05	0.03	0.13
traccar	1	0	1	0	1	0	0.03	0.08
truth	0.95	0.01	0.98	0.02	0.97	0.01	0.39	0.15
twilio-java	0.83	0.02	0.93	0.05	0.88	0.03	0.21	0.15
u2020	0.97	0	0.98	0.01	0.98	0.01	0.22	0.18
unirest-java	0.94	0	0.97	0.01	0.95	0.01	0.01	0.08
waffle	0.99	0	1	0	1	0	0.15	0.23
webcam-capture	0.99	0	1	0	1	0	0.15	0.19
wire	1	0	1	0	1	0	0.03	0.11
xtreemfs	0.97	0.02	0.99	0.02	0.98	0.01	0.47	0.36
yobi	1	0	1	0	1	0	0	0

Table A5. Descriptive Statistics of the Dependent Variables for an RF Model after Applying MF

Project	Precision		Recall		F1		MCC	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD
AcDisplay	0.84	0.05	0.72	0.08	0.77	0.04	0.62	0.06
DDT	0.9	0.02	0.96	0.05	0.93	0.02	0.61	0.04
DSpace	0.99	0.0	1.0	0.0	1.0	0.0	0.8	0.06
HearthSim	1.0	0.0	1.0	0.0	1.0	0.0	0.52	0.19
HikariCP	0.94	0.01	0.97	0.07	0.95	0.03	0.61	0.12
Hydra	0.97	0.01	1.0	0.0	0.98	0.0	0.78	0.05
Hystrix	0.85	0.04	0.9	0.07	0.87	0.03	0.69	0.06
Jest	0.95	0.01	0.99	0.01	0.97	0.01	0.68	0.08

(Continued)

Table A5. Continued

Project	Precision		Recall		F1		MCC	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD
LittleProxy	0.95	0.01	1.0	0.01	0.97	0.01	0.75	0.07
MozStumbler	1.0	0.0	1.0	0.0	1.0	0.0	0.78	0.31
OpenRefine	0.98	0.01	1.0	0.0	0.99	0.01	0.86	0.09
ProjectRed	1.0	0.01	0.97	0.03	0.98	0.02	0.96	0.04
RoaringBitmap	0.99	0.0	0.99	0.02	0.99	0.01	0.62	0.15
Singularity	0.9	0.04	0.96	0.04	0.93	0.02	0.72	0.11
DSpace	0.99	0.0	1.0	0.0	1.0	0.0	0.80	0.06
auto	1.00	0.0	1.0	0.0	1.0	0.0	0.78	0.14
airlift	0.9	0.06	0.85	0.16	0.87	0.11	0.8	0.16
analytics-android	0.98	0.01	1.0	0.0	0.99	0.01	0.85	0.09
android	0.94	0.02	0.97	0.03	0.95	0.02	0.83	0.05
android-maven-plugin	0.98	0.01	1.0	0.01	0.99	0.01	0.73	0.12
assertj-android	0.88	0.03	0.96	0.01	0.92	0.02	0.62	0.11
auto	1.0	0.0	1.0	0.0	1.0	0.0	0.78	0.14
basex	0.99	0.0	1.0	0.01	0.99	0.0	0.72	0.09
blueflood	0.95	0.01	1.0	0.0	0.97	0.0	0.71	0.05
blueprints	0.9	0.05	0.94	0.04	0.92	0.02	0.77	0.07
bnd	0.96	0.01	1.0	0.0	0.98	0.0	0.77	0.03
brightspot-cms	0.94	0.04	0.98	0.02	0.96	0.01	0.86	0.06
cas-addons	1.0	0.0	1.0	0.0	1.0	0.0	0.89	0.14
cassandra-reaper	0.95	0.02	0.99	0.0	0.97	0.01	0.77	0.1
ccw	0.92	0.02	0.98	0.01	0.95	0.01	0.8	0.05
checkstyle	1.0	0.0	1.0	0.0	1.0	0.0	0.81	0.12
cloudify	0.9	0.01	0.98	0.01	0.94	0.01	0.57	0.06
core	1.0	0.0	1.0	0.0	1.0	0.0	0.47	0.2
dagger	0.99	0.01	1.0	0.0	0.99	0.0	0.8	0.15
dropwizard	0.99	0.0	1.0	0.0	0.99	0.0	0.7	0.05
dynjs	0.99	0.01	1.0	0.0	0.99	0.0	0.79	0.13
error-prone	1.0	0.0	1.0	0.0	1.0	0.0	0.81	0.15
frontend-maven-plugin	0.98	0.0	1.0	0.0	0.99	0.0	0.7	0.06
go-lang-idea-plugin	0.99	0.01	1.0	0.0	0.99	0.0	0.8	0.12
goclipse	1.0	0.0	1.0	0.0	1.0	0.0	0.66	0.13
gpslogger	0.98	0.01	1.0	0.0	0.99	0.0	0.79	0.08
hivemall	1.0	0.0	1.0	0.0	1.0	0.0	0.8	0.15
htm.java	1.0	0.0	1.0	0.0	1.0	0.0	0.81	0.21
idea-gitignore	0.9	0.03	0.99	0.02	0.94	0.01	0.52	0.18
jInstagram	0.99	0.0	1.0	0.0	1.0	0.0	0.72	0.1
jPOS	1.0	0.0	1.0	0.0	1.0	0.0	0.76	0.2
jade4j	0.99	0.0	1.0	0.0	0.99	0.0	0.38	0.3
javaslang	1.0	0.0	1.0	0.01	1.0	0.01	0.79	0.2
jcabi-aspects	0.93	0.03	0.99	0.01	0.96	0.01	0.71	0.1
jcabi-github	0.91	0.07	0.97	0.02	0.94	0.05	0.8	0.15

(Continued)

Table A5. Continued

Project	Precision		Recall		F1		MCC	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD
jcabi-http	0.96	0.03	0.99	0.0	0.97	0.01	0.79	0.11
jedis	0.99	0.0	0.99	0.03	0.99	0.01	0.72	0.16
jmeter-plugins	0.98	0.02	0.97	0.05	0.97	0.03	0.85	0.14
jmonkeyengine	1.0	0.0	1.0	0.0	1.0	0.0	0.75	0.13
jmxtrans	0.99	0.0	1.0	0.0	1.0	0.0	0.82	0.1
joda-time	1.0	0.0	1.0	0.0	1.0	0.0	0.73	0.22
jodd	1.0	0.0	1.0	0.0	1.0	0.0	0.74	0.13
jphp	0.95	0.02	1.0	0.0	0.97	0.01	0.73	0.1
jsonld-java	1.0	0.0	1.0	0.0	1.0	0.0	0.76	0.1
jsprit	1.0	0.0	1.0	0.0	1.0	0.0	0.83	0.24
keywhiz	1.0	0.0	1.0	0.0	1.0	0.0	0.5	0.53
lenskit	0.99	0.0	1.0	0.0	1.0	0.0	0.73	0.1
less4j	0.96	0.02	1.0	0.0	0.98	0.01	0.74	0.12
logback	0.99	0.0	1.0	0.0	0.99	0.0	0.74	0.07
lorsource	0.99	0.0	1.0	0.01	0.99	0.0	0.7	0.09
maven-git-commit-id-plugin	0.96	0.01	0.99	0.01	0.98	0.01	0.76	0.06
metrics	0.96	0.05	0.98	0.01	0.97	0.03	0.81	0.19
mockito	0.99	0.01	1.0	0.0	0.99	0.0	0.78	0.1
mybatis-3	1.0	0.0	1.0	0.0	1.0	0.0	0.65	0.24
nokogiri	0.92	0.04	0.94	0.05	0.93	0.03	0.82	0.06
nutz	0.86	0.03	0.94	0.04	0.9	0.02	0.69	0.05
okhttp	0.91	0.02	0.96	0.04	0.93	0.01	0.7	0.05
onebusaway-android	1.0	0.0	1.0	0.0	1.0	0.0	0.77	0.15
openwayback	1.0	0.0	1.0	0.0	1.0	0.0	0.87	0.14
owner	1.0	0.0	1.0	0.0	1.0	0.0	0.73	0.19
p6spy	0.92	0.05	0.91	0.08	0.91	0.03	0.8	0.06
parceler	0.99	0.0	1.0	0.0	1.0	0.0	0.65	0.13
pdfsam	0.89	0.04	0.96	0.04	0.92	0.03	0.53	0.16
play-authenticate	0.98	0.01	1.0	0.01	0.99	0.01	0.8	0.11
psi-probe	1.0	0.0	1.0	0.0	1.0	0.0	0.91	0.17
pushy	1.0	0.0	1.0	0.0	1.0	0.0	0.6	0.52
querydsl	0.99	0.0	1.0	0.0	0.99	0.0	0.71	0.09
quickml	0.99	0.0	1.0	0.0	0.99	0.0	0.72	0.04
qulice	0.99	0.0	1.0	0.0	0.99	0.0	0.72	0.18
restlet-framework-java	0.85	0.11	0.92	0.07	0.88	0.09	0.71	0.23
retrofit	1.0	0.0	1.0	0.0	1.0	0.0	0.88	0.1
rewrite	0.91	0.05	0.96	0.05	0.93	0.04	0.74	0.15
rexster	1.0	0.0	1.0	0.0	1.0	0.0	0.82	0.1
robospice	0.85	0.05	0.76	0.1	0.8	0.06	0.67	0.08
rultor	0.86	0.06	0.94	0.04	0.9	0.03	0.63	0.13
rxjava-jdbc	1.0	0.0	1.0	0.0	1.0	0.0	0.72	0.29
selendroid	0.95	0.06	0.97	0.04	0.96	0.05	0.86	0.16

(Continued)

Table A5. Continued

Project	Precision		Recall		F1		MCC	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD
seyren	0.99	0.01	1.0	0.0	0.99	0.0	0.51	0.37
sms-backup-plus	1.0	0.0	1.0	0.0	1.0	0.0	0.78	0.09
spark	0.99	0.0	1.0	0.0	1.0	0.0	0.71	0.14
spring-cloud-config	0.98	0.01	1.0	0.0	0.99	0.0	0.72	0.08
springside4	0.86	0.05	0.92	0.04	0.89	0.03	0.65	0.1
storio	0.97	0.0	1.0	0.0	0.99	0.0	0.72	0.06
storm	0.86	0.07	0.8	0.1	0.83	0.06	0.72	0.1
structr	0.82	0.1	0.76	0.13	0.78	0.06	0.63	0.1
stubby4j	0.85	0.03	0.96	0.02	0.9	0.02	0.69	0.07
thredds	0.92	0.03	0.97	0.03	0.94	0.02	0.76	0.09
traccar	1.0	0.0	1.0	0.0	1.0	0.0	0.59	0.15
truth	0.97	0.01	1.0	0.0	0.98	0.01	0.72	0.1
twilio-java	0.9	0.02	0.98	0.01	0.94	0.01	0.62	0.05
u2020	0.99	0.01	0.99	0.03	0.99	0.02	0.74	0.19
unirest-java	0.97	0.01	0.98	0.04	0.97	0.02	0.62	0.17
waffle	1.0	0.0	1.0	0.0	1.0	0.0	0.75	0.11
webcam-capture	1.0	0.0	1.0	0.0	1.0	0.0	0.7	0.05
wire	1.0	0.0	1.0	0.0	1.0	0.0	0.6	0.52
xtreemfs	0.99	0.0	1.0	0.01	0.99	0.0	0.84	0.05
yobi	1.0	0.0	1.0	0.0	1.0	0.0	0.73	0.19

Table A6. Descriptive Statistics of the Dependent Variables for an RF Model after Applying CF

Project	Precision		Recall		F1		MCC	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD
AcDisplay	0.7	0.07	0.48	0.13	0.56	0.08	0.34	0.07
DDT	0.85	0.01	0.98	0.01	0.91	0.01	0.39	0.06
DSpace	0.98	0.0	1.0	0.0	0.99	0.0	0.56	0.1
HearthSim	0.99	0.0	1.0	0.0	1.0	0.0	0.31	0.23
HikariCP	0.91	0.02	0.99	0.01	0.95	0.01	0.45	0.12
Hydra	0.95	0.01	1.0	0.0	0.97	0.0	0.54	0.07
Hystrix	0.75	0.09	0.76	0.17	0.74	0.08	0.41	0.07
Jest	0.92	0.01	0.99	0.01	0.95	0.01	0.48	0.08
LittleProxy	0.92	0.01	0.99	0.01	0.95	0.01	0.49	0.06
MozStumbler	1.0	0.0	1.0	0.0	1.0	0.0	0.44	0.48
OpenRefine	0.97	0.02	1.0	0.01	0.98	0.01	0.76	0.15
ProjectRed	0.9	0.09	0.96	0.06	0.92	0.05	0.8	0.15
RoaringBitmap	0.99	0.0	1.0	0.0	0.99	0.0	0.42	0.08
Singularity	0.82	0.05	0.94	0.08	0.87	0.04	0.45	0.15
Dspace	0.98	0.0	1.0	0.0	0.99	0.0	0.56	0.10

(Continued)

Table A6. Continued

Project	Precision		Recall		F1		MCC	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD
auto	0.99	0.0	1.0	0.0	1.00	0.0	0.59	0.15
airlift	0.77	0.19	0.77	0.14	0.76	0.15	0.61	0.26
analytics-android	0.96	0.01	1.0	0.0	0.98	0.0	0.6	0.1
android	0.87	0.05	0.97	0.06	0.92	0.02	0.7	0.09
android-maven-plugin	0.96	0.01	0.98	0.04	0.97	0.02	0.43	0.09
assertj-android	0.81	0.02	0.98	0.01	0.89	0.01	0.4	0.08
auto	0.99	0.0	1.0	0.0	1.0	0.0	0.59	0.15
basex	0.98	0.0	1.0	0.01	0.99	0.0	0.5	0.13
blueflood	0.92	0.01	1.0	0.0	0.96	0.0	0.51	0.06
blueprints	0.83	0.06	0.9	0.07	0.86	0.03	0.6	0.1
bnd	0.93	0.01	1.0	0.0	0.96	0.0	0.56	0.04
brightspot-cms	0.88	0.04	0.96	0.03	0.92	0.02	0.71	0.07
cas-addons	1.0	0.0	1.0	0.0	1.0	0.0	0.72	0.22
cassandra-reaper	0.91	0.02	0.99	0.01	0.95	0.01	0.53	0.14
ccw	0.85	0.02	0.97	0.02	0.91	0.02	0.6	0.08
checkstyle	1.0	0.0	1.0	0.0	1.0	0.0	0.59	0.12
cloudify	0.86	0.01	0.96	0.07	0.9	0.03	0.32	0.08
core	1.0	0.0	1.0	0.0	1.0	0.0	0.31	0.23
dagger	0.98	0.01	1.0	0.0	0.99	0.0	0.64	0.12
dropwizard	0.99	0.0	1.0	0.0	0.99	0.0	0.53	0.09
dynjs	0.98	0.01	1.0	0.0	0.99	0.0	0.59	0.21
error-prone	1.0	0.0	1.0	0.0	1.0	0.0	0.31	0.41
frontend-maven-plugin	0.97	0.0	0.99	0.02	0.98	0.01	0.48	0.1
go-lang-idea-plugin	0.98	0.01	0.99	0.03	0.98	0.01	0.59	0.15
goclipse	1.0	0.0	1.0	0.0	1.0	0.0	0.5	0.15
gpslogger	0.96	0.01	1.0	0.01	0.98	0.01	0.55	0.13
hivemall	1.0	0.0	1.0	0.0	1.0	0.0	0.61	0.14
htm.java	1.0	0.0	1.0	0.0	1.0	0.0	0.68	0.22
idea-gitignore	0.87	0.01	0.99	0.01	0.92	0.01	0.35	0.14
jInstagram	0.99	0.0	1.0	0.0	0.99	0.0	0.53	0.09
jPOS	1.0	0.0	1.0	0.0	1.0	0.0	0.62	0.15
jade4j	0.99	0.0	1.0	0.0	0.99	0.0	0.35	0.23
javaslang	1.0	0.0	1.0	0.0	1.0	0.0	0.65	0.18
jcabi-aspects	0.9	0.03	0.99	0.01	0.94	0.02	0.53	0.17
jcabi-github	0.86	0.08	0.86	0.16	0.86	0.11	0.61	0.25
jcabi-http	0.91	0.01	0.99	0.02	0.95	0.01	0.55	0.09
jedis	0.98	0.0	0.99	0.02	0.99	0.01	0.5	0.1
jmeter-plugins	0.97	0.03	0.99	0.01	0.98	0.01	0.83	0.12
jmonkeyengine	0.99	0.0	1.0	0.0	1.0	0.0	0.6	0.14
jmxtrans	0.99	0.01	1.0	0.0	0.99	0.0	0.56	0.23
joda-time	1.0	0.0	1.0	0.0	1.0	0.0	0.58	0.36
jodd	0.99	0.01	0.81	0.41	0.81	0.39	0.38	0.24

(Continued)

Table A6. Continued

Project	Precision		Recall		F1		MCC	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD
jphp	0.92	0.02	0.99	0.02	0.95	0.01	0.54	0.12
jsonld-java	1.0	0.0	1.0	0.0	1.0	0.0	0.62	0.17
jsprit	1.0	0.0	1.0	0.0	1.0	0.0	0.74	0.21
keywhiz	1.0	0.0	1.0	0.0	1.0	0.0	0.5	0.53
lenskit	0.99	0.0	1.0	0.0	1.0	0.0	0.55	0.09
less4j	0.95	0.02	1.0	0.0	0.97	0.01	0.64	0.17
logback	0.98	0.0	1.0	0.0	0.99	0.0	0.49	0.08
lorsource	0.99	0.0	1.0	0.0	0.99	0.0	0.52	0.17
maven-git-commit-id-plugin	0.92	0.01	0.99	0.01	0.96	0.01	0.49	0.07
metrics	0.9	0.05	0.98	0.01	0.94	0.03	0.6	0.23
mockito	0.98	0.01	0.99	0.02	0.98	0.01	0.55	0.13
mybatis-3	1.0	0.0	1.0	0.0	1.0	0.0	0.5	0.28
nokogiri	0.84	0.06	0.89	0.04	0.86	0.03	0.65	0.1
nutz	0.78	0.04	0.87	0.07	0.82	0.03	0.43	0.09
okhttp	0.84	0.01	0.95	0.06	0.89	0.03	0.49	0.09
onebusaway-android	1.0	0.0	1.0	0.0	1.0	0.0	0.57	0.14
openwayback	1.0	0.0	1.0	0.0	1.0	0.0	0.7	0.19
owner	0.99	0.0	1.0	0.0	1.0	0.0	0.44	0.19
p6spy	0.72	0.12	0.92	0.12	0.8	0.07	0.39	0.29
parceler	0.99	0.0	1.0	0.0	0.99	0.0	0.49	0.16
pdfsam	0.85	0.05	0.94	0.13	0.89	0.05	0.28	0.1
play-authenticate	0.97	0.01	1.0	0.01	0.98	0.01	0.64	0.14
psi-probe	1.0	0.0	1.0	0.0	1.0	0.0	0.81	0.26
pushy	1.0	0.0	1.0	0.0	1.0	0.0	0.2	0.42
querydsl	0.98	0.0	1.0	0.0	0.99	0.0	0.59	0.08
quickml	0.98	0.0	1.0	0.0	0.99	0.0	0.51	0.12
qulice	0.99	0.0	1.0	0.0	0.99	0.0	0.61	0.17
restlet-framework-java	0.8	0.13	0.64	0.25	0.67	0.14	0.42	0.17
retrofit	0.99	0.0	1.0	0.0	0.99	0.0	0.62	0.11
rewrite	0.82	0.04	0.89	0.12	0.85	0.05	0.43	0.13
rexster	0.99	0.0	1.0	0.0	1.0	0.0	0.61	0.15
robospice	0.69	0.07	0.62	0.14	0.64	0.09	0.41	0.12
rultor	0.8	0.06	0.89	0.09	0.83	0.04	0.41	0.14
rxjava-jdbc	1.0	0.0	1.0	0.0	1.0	0.0	0.52	0.37
selendroid	0.82	0.06	0.95	0.02	0.88	0.03	0.55	0.14
seyren	0.98	0.01	1.0	0.0	0.99	0.0	0.27	0.32
sms-backup-plus	0.99	0.0	1.0	0.0	1.0	0.0	0.66	0.18
spark	0.99	0.0	1.0	0.0	0.99	0.0	0.42	0.22
spring-cloud-config	0.96	0.01	1.0	0.0	0.98	0.0	0.51	0.15
springside4	0.78	0.04	0.83	0.19	0.79	0.11	0.42	0.12
storio	0.96	0.01	1.0	0.0	0.98	0.0	0.47	0.09
storm	0.73	0.1	0.67	0.19	0.68	0.11	0.49	0.14

(Continued)

Table A6. Continued

Project	Precision		Recall		F1		MCC	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD
structr	0.65	0.12	0.66	0.23	0.62	0.1	0.36	0.1
stubby4j	0.77	0.03	0.92	0.08	0.83	0.03	0.47	0.08
thredds	0.85	0.03	0.96	0.03	0.9	0.02	0.56	0.11
traccar	1.0	0.0	1.0	0.0	1.0	0.0	0.41	0.18
truth	0.95	0.01	1.0	0.0	0.98	0.0	0.57	0.1
twilio-java	0.84	0.01	0.99	0.01	0.91	0.01	0.38	0.06
u2020	0.98	0.01	1.0	0.0	0.99	0.0	0.48	0.21
unirest-java	0.95	0.01	1.0	0.0	0.98	0.01	0.49	0.15
waffle	0.99	0.0	1.0	0.0	1.0	0.0	0.45	0.22
webcam-capture	1.0	0.0	1.0	0.0	1.0	0.0	0.56	0.09
wire	1.0	0.0	1.0	0.0	1.0	0.0	0.6	0.52
xtreemfs	0.98	0.01	0.99	0.01	0.99	0.0	0.69	0.14
yobi	1.0	0.0	1.0	0.0	1.0	0.0	0.62	0.16

Table A7. The Distribution of Build Outcome Data Points within the Analyzed Subjects

Project	class 0	class 1	class ratio
AcDisplay	19,424	15,461	44.319
DDT	11,929	51,879	18.695
DSpace	2,330	83,544	2.713
HearthSim	400	66,046	0.601
HikariCP	3,952	28,938	12.015
Hydra	623	7,270	7.893
Hystrix	18,558	24,944	42.66
Jest	2,862	22,863	11.125
LittleProxy	1,025	7,737	11.698
MozStumbler	15	8,625	0.173
OpenRefine	357	3,814	8.559
ProjectRed	450	702	39.062
RoaringBitmap	788	41,989	1.842
Singularity	2,267	6,257	26.595
airlift	14,632	9,380	39.063
analytics-android	507	6,926	6.820
android	4,587	11,046	29.341
android-maven-plugin	4,598	80,031	5.433
assertj-android	3,798	12,111	23.873
auto	137	12,917	1.049
basex	2,030	57,303	3.421
blueflood	4,886	40,437	10.78
blueprints	16,890	29,439	36.456
bnd	3,852	31,838	10.7929

(Continued)

Table A7. Continued

Project	class0	class1	class_ratio
brightspot-cms	2,981	7,040	29.747
cas-addons	66	8,049	0.813
cassandra-reaper	1,082	7,324	12.871
ccw	4,070	11,538	26.076
checkstyle	370	103,059	0.357
cloudify	56,428	265,933	17.504
core	420	84,118	0.496
dagger	119	3,355	3.425
dropwizard	1,154	54,038	2.090
dynjs	1,033	35,286	2.844
error-prone	26	101,295	0.025
frontend-maven-plugin	110	2,774	3.814
go-lang-idea-plugin	1,319	32,505	3.899
goclipse	509	76,552	0.66
gpslogger	1,009	15,405	6.147
hivemall	169	22,752	0.737
htm.java	470	53,930	0.863
idea-gitignore	4,641	24,359	16.003
jInstagram	294	20,225	1.432
jPOS	160	38,231	0.416
jade4j	283	16,171	1.719
java-design-patterns	55	23,249	0.236
javaslang	1,151	193,409	0.591
jcabi-aspects	904	4,669	16.221
jcabi-github	5,937	11,839	33.399
jcabi-http	624	3,913	13.753
jedis	984	40,365	2.379
jinjava	8	12,747	0.062
jmeter-plugins	8,995	56,751	13.681
jmonkeyengine	928	77,326	1.185
jmxtrans	198	8,497	2.277
joda-time	37	19,276	0.191
jodd	1,546	153,410	0.997
jphp	20,586	141,821	12.675
jsonld-java	78	34,127	0.228
jsonschema2pojo	3	15,091	0.0198
jsprit	118	23,640	0.496
keywhiz	5	12,791	0.039
lenskit	848	58,210	1.435
less4j	8,259	73,837	10.060
logback	2,046	71,501	2.781
lorsource	751	34,964	2.102
maven-git-commit-id-plugin	1,714	14,099	10.839
metrics	1,778	7,857	18.453

(Continued)

Table A7. Continued

Project	class0	class1	class_ratio
mockito	2,976	71,665	3.987
mybatis-3	604	105,989	0.566
nodeclipse-1	27	23,959	0.112
nokogiri	10,368	15,481	40.109
nutz	22,390	42,002	34.771
okhttp	17,121	53,828	24.1314
onebusaway-android	66	22,335	0.294
openwayback	31	12,905	0.239
owner	266	23,163	1.135
p6spy	6,353	8,975	41.447
parceler	284	17,457	1.6
pdfsam	24,027	106,294	18.436
picard	378	11,017	3.317
play-authenticate	330	5,229	5.936
psi-probe	100	57,045	0.174
pushy	9	4,705	0.19
querydsl	929	38,383	2.363
quickml	486	14,942	3.15
qulice	253	11,216	2.205
restlet-framework-java	52,947	65,339	44.761
retrofit	358	18,901	1.858
rewrite	4,091	10,717	27.627
rexster	515	37,728	1.346
robospice	8,475	6,487	43.356
rultor	10,632	23,676	30.989
rxjava-jdbc	20	8,677	0.229
selendroid	18,702	42,747	30.435
seyren	166	8,447	1.927
sms-backup-plus	215	16,358	1.297
spark	94	6,345	1.459
spring-cloud-config	1,368	23,983	5.396
springside4	7,317	14,330	33.801
storio	911	14,309	5.985
storm	18,316	12,936	41.392
structr	73,595	57,331	43.788
stubby4j	14,505	27,490	34.539
thredds	6,920	20,015	25.691
traccar	144	76,165	0.188
truth	1,351	17,931	7.006
twilio-java	6,285	25,939	19.504
u2020	149	4,729	3.0545
unirest-java	262	3,803	6.445
waffle	168	20,320	0.8199

(Continued)

Table A7. Continued

Project	class0	class1	class_ratio
webcam-capture	267	38,456	0.689
wire	11	52,079	0.0211
xtreemfs	2,494	53,452	4.457
yobi	106	23,134	0.4561

Table A8. The Mean Performance Scores of a Model Trained on Codebert Embeddings for Build Outcome Prediction

Project	Algorithm	Precision	Recall	F1-score	MCC
HikariCP	CF	0.89	0.92	0.90	0.14
HikariCP	DB	0.87	0.91	0.89	−0.08
HikariCP	MF	0.91	0.96	0.93	0.29
HikariCP	None	0.88	0.98	0.93	0.07
OpenRefine	CF	0.93	1.00	0.96	0.28
OpenRefine	DB	0.91	0.83	0.87	−0.03
OpenRefine	MF	0.99	0.98	0.98	0.85
OpenRefine	None	0.95	0.98	0.96	0.45
ProjectRed	CF	0.93	0.93	0.93	0.82
ProjectRed	DB	0.89	0.79	0.82	0.63
ProjectRed	MF	1.00	0.96	0.98	0.95
ProjectRed	None	0.86	0.88	0.87	0.66
android	CF	0.86	0.95	0.90	0.63
android	DB	0.90	0.88	0.89	0.64
android	MF	0.93	0.95	0.94	0.78
android	None	0.89	0.95	0.92	0.69
brightspot	CF	0.89	0.82	0.85	0.55
brightspot	DB	0.87	0.90	0.88	0.59
brightspot	MF	0.95	0.98	0.96	0.87
brightspot	None	0.84	0.92	0.88	0.52
cloudify	CF	0.85	0.90	0.88	0.22
cloudify	DB	0.83	0.69	0.75	0.02
cloudify	MF	0.92	0.95	0.93	0.60
cloudify	None	0.83	0.98	0.90	0.04
core	CF	1.00	0.93	0.96	0.16
core	DB	1.00	0.62	0.77	0.02
core	MF	1.00	0.98	0.99	0.41
core	None	1.00	1.00	1.00	0.10
dropwizard	CF	0.98	0.92	0.95	0.21
dropwizard	DB	0.98	0.79	0.88	0.01
dropwizard	MF	0.98	0.98	0.98	0.27
dropwizard	None	0.98	1.00	0.99	0.10
pdfsam	CF	0.85	0.86	0.84	0.19
pdfsam	DB	0.84	0.87	0.85	0.17

(Continued)

Table A8. Continued

Project	Algorithm	Precision	Recall	F1-score	MCC
pdfsam	MF	0.90	0.90	0.89	0.49
pdfsam	None	0.82	0.98	0.89	0.01

Table A9. The Mean Performance Scores of a Model Trained on BoW Feature Vectors for Build Outcome Prediction

Project	Algorithm	Precision	Recall	F1-score	MCC
HikariCP	None	0.89	0.98	0.93	0.09
HikariCP	CF	0.91	0.99	0.95	0.45
HikariCP	DB	0.88	0.95	0.91	−0.03
HikariCP	MF	0.94	0.97	0.95	0.61
OpenRefine	None	0.96	0.90	0.93	0.40
OpenRefine	CF	0.96	0.98	0.97	0.55
OpenRefine	DB	0.91	0.84	0.87	−0.02
OpenRefine	MF	0.98	1.00	0.99	0.86
ProjectRed	None	0.90	0.84	0.85	0.68
ProjectRed	CF	0.90	0.96	0.92	0.80
ProjectRed	DB	0.91	0.80	0.84	0.66
ProjectRed	MF	0.89	0.86	0.86	0.69
android	None	0.88	0.95	0.92	0.68
android	CF	0.87	0.97	0.92	0.70
android	DB	0.95	0.83	0.88	0.66
android	MF	0.94	0.97	0.95	0.83
brightspot	None	0.84	0.90	0.87	0.50
brightspot	CF	0.88	0.96	0.92	0.71
brightspot	DB	0.82	0.84	0.82	0.37
brightspot	MF	0.94	0.98	0.96	0.86
cloudify	None	0.83	0.95	0.88	0.01
cloudify	CF	0.86	0.96	0.90	0.32
cloudify	DB	0.84	0.57	0.68	0.04
cloudify	MF	0.90	0.98	0.94	0.57
core	None	1.00	1.00	1.00	0.07
core	CF	1.00	1.00	1.00	0.31
core	DB	1.00	0.68	0.81	0.02
core	MF	1.00	1.00	1.00	0.47
dropwizard	None	0.98	1.00	0.99	0.11
dropwizard	CF	0.98	0.99	0.99	0.06
dropwizard	DB	0.98	0.99	0.98	0.01
dropwizard	MF	0.99	1.00	0.99	0.70
pdfsam	None	0.81	0.62	0.70	0.21
pdfsam	CF	0.85	0.94	0.89	0.28
pdfsam	DB	0.82	0.82	0.81	0.02
pdfsam	MF	0.89	0.96	0.92	0.53

Received 17 June 2023; revised 15 May 2025; accepted 5 August 2025