applied sciences

MDPI

Article

Making More with Less: Improving Software Testing Outcomes Using a Cross-Project and Cross-Language ML Classifier Based on Cost-Sensitive Training

Alexandre M. Nascimento [1,2,*], Gabriel Kenji G. Shimanuki [3] and Luiz Alberto V. Dias [2]

[1] Mechanical Engineering Department, Stanford University, Stanford, CA 94305, USA
[2] Instituto Tecnológico de Aeronáutica (ITA), São José dos Campos 12228-900, Brazil; vdias@ita.br
[3] Department of Computer Engineering and Digital Systems, Escola Politécnica, Universidade de São Paulo (USP), São Paulo 05508-220, Brazil; gabrielshimanuki@usp.br
* Correspondence: alexandremoreiranascimento@alum.mit.edu

Featured Application: The technique uses Machine Learning (ML) models to support decision-making on software testing scope and resource allocation to augment the outcomes with the available resources.

Abstract: As digitalization expands across all sectors, the economic toll of software defects on the U.S. economy reaches up to $2.41 trillion annually. High-profile incidents like the Boeing 787-Max 8 crash have shown the devastating potential of these defects, highlighting the critical importance of software testing within quality assurance frameworks. However, due to its complexity and resource intensity, the exhaustive nature of comprehensive testing often surpasses budget constraints. This research utilizes a machine learning (ML) model to enhance software testing decisions by pinpointing areas most susceptible to defects and optimizing scarce resource allocation. Previous studies have shown promising results using cost-sensitive training to refine ML models, improving predictive accuracy by reducing false negatives through addressing class imbalances in defect prediction datasets. This approach facilitates more targeted and effective testing efforts. Nevertheless, these models' in-company generalizability across different projects (cross-project) and programming languages (cross-language) remained untested. This study validates the approach's applicability across diverse development environments by integrating various datasets from distinct projects into a unified dataset, using a more interpretable ML technique. The results demonstrate that ML can support software testing decisions, enabling teams to identify up to 7× more defective modules compared to benchmark with the same testing effort.

Keywords: machine learning; imbalance; software defect prediction; NASA MDP; random forest; software quality; generalization; cost-sensitive; cross-language; cross-project

check for updates

Citation: Nascimento, A.M.; Shimanuki, G.K.G.; Dias, L.A.V. Making More with Less: Improving Software Testing Outcomes Using a Cross-Project and Cross-Language ML Classifier Based on Cost-Sensitive Training. *Appl. Sci.* **2024**, *14*, 4880. https://doi.org/10.3390/app14114880

Academic Editor: Andrea Prati

Received: 14 April 2024
Revised: 21 May 2024
Accepted: 30 May 2024
Published: 4 June 2024

1. Introduction

Over the last decades, society has been experiencing growth in digitalization in practically all professional activities. As economic activities become more dependent on software, the impact of software quality issues increases. Studies have estimated the annual cost of software bugs to the US economy from $59.5 billion to $2.41 trillion [1,2], which means the per capita yearly cost of software issues is $7230.9. In fact, software malfunctions have been playing an essential role in accidents damaging the reputation and market value of traditional companies, such as the example of the Maneuvering Characteristics Augmentation System (MCAS) in the Boeing 787-Max 8 case [3], resulting in a $29Bi market value loss in a few days [4] and taking over 350 human lives [5]. Thus, it is possible to say that software quality assurance plays a pivotal role in the US economy. Since software testing is one of the core activities in software quality assurance [6], ultimately, it plays a crucial role in the US economy.

However, testing every potential software condition is an unattainable task [7–9]. Despite the resources available to be invested in, it is impossible to test all possible software conditions [10,11] since it could take millions of years [12], making the activity useless. On the one hand, because of its complexity, software testing consumes a considerable fraction of software development projects. In fact, it is estimated that up to 50% of the total budget is consumed by the testing activity [13]. On the other hand, the resources available for software testing are usually very scarce [14–17].

As a result, software testing planning requires challenging decision-making to balance conflicting variables (scope size, test coverage, and resource allocation) to obtain most of the effort. Managers must be able to plan the activity to cover the software as much as possible [13]. At the same time, they must be able to reduce the test scope safely [18]. Finally, they need to have the capability to allocate the available resources wisely (testers, tools, and time) [19] to test the software.

Machine learning (ML) models can help managers make better-informed decisions about optimizing the outcomes of a software testing effort, given the availability of resources. A commonly utilized protocol in software defect prediction research is illustrated in Figure 1. ML classifier models are trained to highlight the system modules most prone to defects [20–22], using a historical dataset containing each module's static source-code metrics and an indication of whether it was defective or not [23–27]. By knowing which software modules have higher defect risks, managers can reduce the software testing scope around them and assign the available resources to concentrate their efforts on a more focused approach.

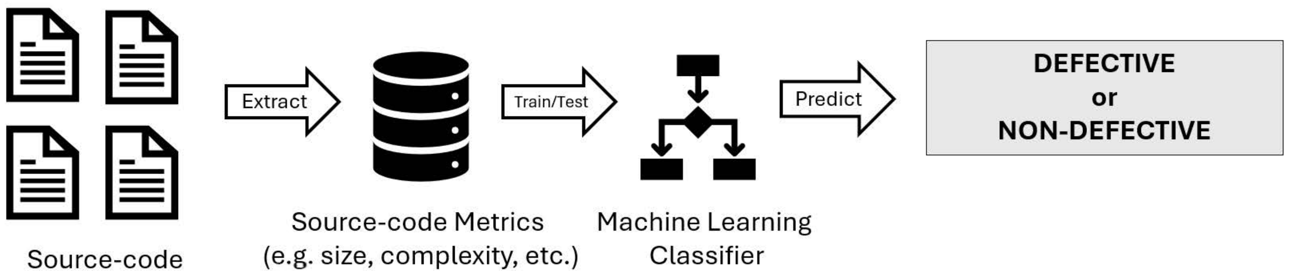


Figure 1. Typical ML training process to create a model to predict defective software modules.

An extensive body of research on software defect prediction based on ML models exists. The literature approaches explore defective prediction models from many perspectives [28–37]. One of the most well-known datasets used in many of those studies is the NASA MDP open datasets [38,39]. Because of their popularity and frequency, they have been used as a common ground to establish a benchmark to support performance comparison among distinct studies. Many studies compare the performance of distinct ML algorithms. Ref. [29] concluded that the utilization of dagging-based classifiers enhanced software defect prediction models relative to baseline classifiers like Naïve Bayes (NB), Decision Tree (DT), and k-nearest neighbor (kNN). Ref. [35] conducted a study comparing Extreme Learning Machine (ELM) and Support Vector Machine (SVM), finding that ELM exhibited superior performance, boosting accuracy from 78.68% to 84.61%. ELMs can be understood as a fast supervised learning algorithm for ANNs, in which input weights are randomly assigned, and output weights are analytically calculated. Finally, the authors suggest a future research direction involving the application of unsupervised and semi-supervised learning algorithms, considering that most investigations have focused on supervised learning. Ref. [34] proposes a new method based on Convolutional Neural Networks (CNNs) to identify patterns associated with software defects. Despite the good results presented, it is shown that they are too sensitive to hyperparameters and scope boundaries (focused on an individual version or project) [34]. Therefore, ML techniques with prediction performances that are less dependent on hyperparameter optimization are valuable in this domain because they empower software testers to become less dependent on ML specialists. Finally, an important aspect related to those studies was pointed out by [29]. According to [29], the perceived efficacy of ML algorithms can vary depending on the performance metrics employed and the specific conditions of the experiment. That issue seems to be related to the use of generic data science metrics for performance evaluation rather than ones more specific to the context of the field domain.

Another way to approach the problem is to consider it from the perspective of feature selection (FS). In [33], an FS approach is presented, utilizing the island binary moth flame combined with SVM, NB, and kNN. Ref. [31] proposed a rank aggregation-based Multi-Filter Selection Method, outperforming traditional methods by increasing their prediction accuracies. For example, it increased NBs' accuracy from 76.33% to 81–82% and DTs' from 83.01% to almost 85%. Furthermore, the study suggests that future research should broaden the study's scope to encompass a wider range of prediction models. In [30,32], both studies demonstrate that the effectiveness of FS methods is influenced by factors such as choice of classifiers, evaluation metrics, and dataset. While FS enhances predictive performance, its efficacy varies across datasets and models, possibly due to class imbalance. While [30] employs only NB and DT, ref. [32] also utilizes kNN and Kernel Logistic Regression (KLR).

Attempting to address the issue from another perspective, some studies focus on the quality of the data that the models utilize. [37] proposes a resampling method utilizing NB, but it fails to outperform benchmarks across all datasets, highlighting that there are no universally effective imbalance learning methods; thus, selecting appropriate methods is crucial. Ref. [36] proposes a method and compares it with existing ones, addressing issues with imbalanced learning, such as interference with real data caused by using SMOTE, emphasizing the importance of focusing on the data quality of synthetic data. Lastly, in addressing the imbalance problem, ref. [28] explores the utilization of Generative Adversarial Networks (GANs) for balancing datasets through synthetic sampling of the minority class. Empirical evidence suggests that GANs demonstrate superior performance compared to traditional methods such as SMOTE. However, it is important to note that undersampling techniques with GANs may result in a degradation in prediction performance due to eliminating crucial samples. Moreover, the authors highlight the potential challenges associated with hyperparameter optimization in GAN-based methods and its impact on the final predictive performance of models.

Indeed, a critical aspect of those datasets used for training ML model to predict defective modules, including NASA's, is their imbalance. That is because defective modules are expected to be a small ratio of the system. Thus, since the ratio of dataset instances with defective modules is usually much smaller than the non-defective ones, the class imbalance becomes a natural characteristic of those datasets. Proper ML model training compensating for the imbalance with one of the existing techniques must be used. However, among the many limitations already pointed out by the literature [39] many studies did not account for the imbalance of the dataset used to induce the ML models [40]. Consequently, reported results are biased towards the majority class (non-defective), resulting in high accuracy levels that hide the ML classifier's actual performance. That unreal information supports poor decision-making for software testing because they usually classify many defective modules as non-defective. Those false negatives (FN) create wrong expectations and optimism about an unreal software high-quality level, misleading managers to lower the software testing efforts and deflecting the efforts from those many misclassified defective modules. Consequently, those issues remain in the software, resulting in future operational failures that could lead to severe consequences.

Studies proposed novel techniques to enhance the learning of the ML model. For example, research [22] demonstrated better ML classifiers for predicting defective software modules using a novel automatic feature engineering approach to create new features that enabled superior information gain in the ML learning process. However, studies relying on that strategy tackled only one aspect of the existing issues. Their ML models were superior at indicating defective software modules. Nevertheless, that optimization ignored vital decision-making information: available resources. Ignoring it reduces their practical utility in actual software testing decision-making since they may suggest a scope that can either not be afforded by or underuse the available resources.

Previous research proposed and evaluated a novel method to support managers in making better decisions to optimize the outcomes of a software testing effort. The method leveraged the dataset imbalance and cost-sensitive ML training to improve the ML model results, considering resource availability and smoothing unwanted FN effects [40]. Ref. [40] demonstrated that the method could improve the prediction of defective software modules in imbalanced datasets. By adjusting the costs (penalty) imposed on FN, the technique has been shown to support decision-making on software testing scope while considering resource availability. Nonetheless, the ML model was tested with unseen data derived from the same single project dataset it used for learning. Although a cross-validation strategy was used, the study did not investigate the technique's generalizability in cross-project and cross-language scenarios inside the same organization (in-organization).

ML generalizability refers to a model's ability to effectively apply what it has learned from the training data to a new context. Developing models that can generalize is a core goal in ML because it directly impacts a model's practical usefulness. A model that generalizes well can accurately interpret and predict outcomes in real-world new situations, highlighting its adaptability and robustness. This is particularly significant in fields like the one studied here, where ML models must adapt to diverse software projects, teams, architectures, and programming languages to be useful. Models with low generalizability perform well on training data but poorly on real-world data, resulting in potentially severe implications in safety-critical applications [41–43].

In the present domain, in-organization generalizability plays an important role when a new software system development project begins without a considerable system defect track record. The lack of a considerable dataset makes it hard for managers to use ML models to get insights about the software testing scope on which their resources should be focused. If no data is available, there is not much workaround. Therefore, a cross-project and cross-language generalizable ML model within the same organization could be a game changer. That ML model, trained with data from other systems (previously developed or under simultaneous development) based on other programming languages, would support managers in making decisions on software testing scope and resource allocation from the initial software development iterations. That would enhance the practical usefulness of those ML model-based techniques.

Notably, synthetic data can be used if little data is available, as in [44]. However, synthetic data has several disadvantages in this domain. First, synthetic data might not accurately reflect the complexities and nuances of real-world software development projects in the organization, especially if little and no broader representative data is available to seed synthetic data generation [45]. This lack of realism can lead to an ML classifier that performs well on synthetic data but poorly on actual project data. Second, since synthetic data is artificially created, there might be skepticism about its validity and reliability, impacting the trust in the predictions made by the ML classifier trained on such data, especially when they challenge the software testing team members' expectations or guesses [46]. Third, creating high-quality synthetic data that accurately mimic real-world scenarios can be challenging and requires expertise in topics rarely present in software testing teams, which can raise the ML-based approach adoption barriers [47]. Finally, poorly generated synthetic data can lead to inaccurate predictions and poor model performance [48], which can result in disastrous consequences in safety-critical software applications.

Therefore, an ML model trained with data from previous or simultaneously under-development projects within the same organization would be more beneficial if in-organization generalizability is demonstrated. This approach leverages real historical data's realism and relevance, ensuring the model is influenced by source-code metrics resulting from the specific processes, team members, tools, and environments used within the organization. It also allows for a broader leveraging of proven patterns and trends observed in past projects, potentially leading to more accurate and reliable predictions. While synthetic data can be a valuable supplement when little data is available, real historical data from the same organization could potentially provide a more robust foundation for training ML classifiers in the present domain if in-organization generalizability is demonstrated.

Another limitation of the original study [40] is that it was validated only with a single ML technique, the artificial neural network (ANN), which has several disadvantages in this problem domain. ANNs require large amounts of data, considerable training times, and suitable hardware due to their high computational cost, which may not be available [49–51]. They also require more challenging data preprocessing, feature engineering, and hyperparameter tuning, which may require a specialization beyond what can be expected from the conventional software testing staff [52,53]. Furthermore, they tend to overfit, especially when the model is too complex relative to the amount and diversity of the training data, leading to poor generalization in new contexts, which is highly undesired in the domain investigated [54,55]. Finally, its black-box nature results in poor explainability and interpretability [56,57]. The lack of explainability and interpretability prevents managers from getting additional information about root causes linked to classifying a module as defective, which could support proactive actions to improve the development teams continuously. Thus, a gap exists in evaluating the cost-sensitive approach using lighter, easier-to-use, and more explainable and interpretable ML techniques.

In this context, the present study aims to tackle those limitations to validate the potential of the cost-sensitive approach to identify the software testing scope while accounting for resource availability. A distinct, computationally lighter, and easier-to-use ML technique with better explainability and interpretability was used on an assembled dataset combining distinct software development projects within the same organization based on different programming languages. Furthermore, the present work expanded the investigation, using a dataset almost 4.5 times larger than that of the baseline study [40]. To our knowledge, no other study has used the proposed approach in the defect prediction domain and validated its potential in-organization generalizability in the way executed here. Finally, performance evaluation metrics that are more appropriate to the research domain proposed in the previous study are refined, and new ones are proposed.

This study is organized into five sections. Section 2 presents the methodology used to support the study's goal. Section 3 presents the experimental results. Section 4 presents the discussion. Finally, Section 5 presents the final remarks and conclusions of the present study.

2. Materials and Methods

This section contains the experimental protocol and materials used to support the research. First, the dataset used for training and evaluating the ML model is described. Then, a brief overview of the ML technique used to induce the ML model is presented. Right after, the experimental protocol used to evaluate the experiment is explained. Then, after the metrics used to evaluate the performance are detailed, a brief description of the statistical tests performed is presented.

2.1. Dataset

Some requirements were considered in the dataset selection for this study. A dataset containing information about the defect incidence in software modules from multiple projects based on distinct programming languages in the same organization was required to support the achievement of the paper's goals. Moreover, a well-known and widely used dataset in the literature of the present research domain was necessary because of the established references and benchmarks to support future comparisons. Finally, since the present study's results are compared to the previous one, using datasets from the same organization was highly desired.

NASA opened 14 datasets regarding distinct software development projects to support research on software module defect prediction [58]. The datasets cover 14 distinct software development projects based mainly on three programming languages: C, C++, and Java. The nature of those software systems from the same organization was also different, probably with distinct architectures and design patterns, since some were related to, for example, spacecraft instruments (CM1), a storage management system for receiving and processing ground data (KC1, KC2, and KC3), a combustion experiment of a space shuttle (MC1), a video guidance system (MC2), a zero-gravity experiment related to combustion (MW1), flight software from an earth-orbiting satellite (PC1, PC2, and PC4), s dynamic simulator for attitude control systems (PC2), and a cockpit security increase system (PC5) [59,60]. Finally, the original study used a dataset from a single NASA project, which the present one seeks to extend. Therefore, the NASA MDP datasets were a natural choice since they meet all the study's requirements.

Each instance of NASA MDP datasets corresponds to a software module's diverse static source-code metrics (features) and a class indicating whether the module was found to be defective or not. Those source-code metrics characterize code features associated with software quality: distinct lines of code measures, McCabe metrics, Halstead's base, derived measures, and branch count metrics [61–64]. The number of features in each dataset varies slightly, with some having additional source-code metrics compared to others. Moreover, each dataset's number of instances is distinct because of each project's different number of modules. Since NASA MDP datasets became popular, slightly different versions have been available in distinct repositories. The pre-cleaned version [39] was used in the present study. Table 1 shows each NASA dataset's characteristics.

Table 1. A map of dataset characteristics and their features source-code static metrics for supporting cross–language and cross–project merging. A bold X with a dark gray background indicates the feature is present in all the datasets.

	Merged Datasets (NASA Project Name)											
Dataset Characteristics	**CM1**	**JM1**	**KC1**	**KC3**	**MC1**	**MC2**	**MW1**	**PC1**	**PC2**	**PC3**	**PC4**	**PC5**
Number of Instances (Modules)	344	9591	2095	200	8737	125	263	735	1493	1099	1379	16962
Number of Features	37	21	21	39	38	39	37	37	36	37	37	38
Programming Language	C	C	C++	Java	C/C++	C	C	C	C	C	C	C
Feature Name	Common features											
LOC_BLANK	x	x	x	x	x	x	x	x		x	x	x
BRANCH_COUNT	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**
CALL_PAIRS	x			x	x	x	x	x	x	x	x	x
LOC_CODE_AND_COMMENT	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**
LOC_COMMENTS	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**
CONDITION_COUNT	x			x	x	x	x	x	x	x	x	x
CYCLOMATIC_COMPLEXITY	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**
CYCLOMATIC_DENSITY	x			x	x	x	x	x	x	x	x	x
DECISION_COUNT	x			x	x	x	x	x	x	x	x	x
DECISION_DENSITY	x			x		x	x	x	x	x	x	
DESIGN_COMPLEXITY	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**
DESIGN_DENSITY	x			x	x	x	x	x	x	x	x	x
EDGE_COUNT	x			x	x	x	x	x	x	x	x	x
ESSENTIAL_COMPLEXITY	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**
ESSENTIAL_DENSITY	x			x	x	x	x	x	x	x	x	x
LOC_EXECUTABLE	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**
PARAMETER_COUNT	x			x	x	x	x	x	x	x	x	x
HALSTEAD_CONTENT	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**
HALSTEAD_DIFFICULTY	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**
HALSTEAD_EFFORT	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**
HALSTEAD_ERROR_EST	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**
HALSTEAD_LENGTH	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**
HALSTEAD_LEVEL	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**
HALSTEAD_PROG_TIME	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**
HALSTEAD_VOLUME	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**
MAINTENANCE_SEVERITY	x			x	x	x	x	x	x	x	x	x

Table 1. *Cont.*

	Merged Datasets (NASA Project Name)											
Dataset Characteristics	**CM1**	**JM1**	**KC1**	**KC3**	**MC1**	**MC2**	**MW1**	**PC1**	**PC2**	**PC3**	**PC4**	**PC5**
MODIFIED_CONDITION_COUNT	x			x	x	x	x	x	x	x	x	x
MULTIPLE_CONDITION_COUNT	x			x	x	x	x	x	x	x	x	x
NODE_COUNT	x			x	x	x	x	x	x	x	x	x
NORMALIZED_CYLOMATIC_ COMPLEXITY	x			x	x	x	x	x	x	x	x	x
NUM_OPERANDS	X	X	X	X	X	X	X	X	X	X	X	X
NUM_OPERATORS	X	X	X	X	X	X	X	X	X	X	X	X
NUM_UNIQUE_OPERANDS	X	X	X	X	X	X	X	X	X	X	X	X
NUM_UNIQUE_OPERATORS	X	X	X	X	X	X	X	X	X	X	X	X
NUMBER_OF_LINES	x			x	x	x	x	x	x	x	x	x
PERCENT_COMMENTS	x			x	x	x	x	x	x	x	x	x
LOC_TOTAL	X	X	X	X	X	X	X	X	X	X	X	X
GLOBAL_DATA_COMPLEXITY				x	x	x						x
GLOBAL_DATA_DENSITY				x	x	x						x

Based on them, an assembled dataset was used to support the investigation of the cost-sensitive approach [40] regarding cross-project and cross-language in-organization generalizability. Its assembly was done by carefully merging the NASA MDP datasets. From the original NASA database, KC2 and KC4 were excluded due to significant feature discrepancies that could jeopardize the experiments. Thus, 12 NASA MDP datasets remained to be merged. However, the slight difference in each dataset's number and type of features imposes some challenges in this merging process, since it cannot be done by simply concatenating all the files into a single one. That could be one of the reasons for the existing literature gap, since it may have prevented the exploration of this repository's full potential to investigate in-company cross-project and cross-language generalizability.

When multiple datasets have some features in common and others distinct, such as NASA MDP datasets, a proper dataset merging strategy must first identify all the features present in all the datasets (features in common). Then, all the other features that are not contained in at least one dataset must be removed from all the datasets they are contained in. Finally, with all the datasets containing the same number and type of features, all their contents can be copied into a single new file. That process makes all the datasets compatible by ensuring they contain only features in common among all of them before they are merged.

Therefore, first, all the distinct features from all the 12 NASA MDP datasets were identified, as shown in Table 1. Then, each dataset containing each one of those 39 distinct features was identified (marked with "x" in a light gray background cell in Table 1). Right after, all the features contained in the 12 NASA MDP datasets were identified (i.e., with all the corresponding cells marked with "x" in a light gray background cell in Table 1). As a result, a total of 20 features were identified as present in all datasets (marked with "X" in a darker gray background in Table 1). All the other 19 features were removed from the datasets containing them. Finally, all the now compatible 12 NASA MDP datasets were merged into a new single file. The "NASA MDP CROSS-PROJECTS DATASET" was the resulting single cross-project and cross-language dataset, which was used to support the present study and will be opened to the research community for future investigations.

While the dataset used in the original study had 9593 instances with 21 features of a single project's software modules in C language, the assembled dataset contains 43,023 instances and 20 features based on source-code static metrics corresponding to software modules in C, C++, and Java, with no missing values. As expected, the assembled dataset is imbalanced because only 7.4% of modules were defective. Although that imbalance is more aggressive than the original study's (18.33% of the classes defective), no technique, such as oversampling [65], under-sampling [66,67], case weighting [68], or synthetic minority oversampling technique (SMOTE) [69], was used to balance the dataset classes to follow the same protocol used by the original research.

2.2. Machine Learning Framework and Technique

Although advanced cutting-edge ML techniques with excellent hyperparameter optimization running on sophisticated ML frameworks requiring coding skills could potentially achieve superior performance, they reduce the intention of software testing professionals to adopt AI-based solutions by negatively impacting some of the factors influencing their intention to do so: facilitating conditions, expected effort, and self-efficacy [70]. Since software testers and their managers are not likely to be experts in ML, using a codeless ML framework and a more straightforward ML technique that could achieve good performance even without hyperparameter optimization (i.e., using default settings) were requirements.

Therefore, the ML framework used was WEKA (Waikato Environment for Knowledge Analysis) version 3.8.6 [71]. WEKA is a popular, versatile, and accessible open-source ML and data-mining framework developed at the University of Waikato in New Zealand. It provides a Graphical User Interface for a codeless environment encompassing a collection of visualization tools and algorithms for data analysis and predictive modeling for anyone interested in data mining and machine learning, from beginners to advanced practitioners.

Regarding the ML technique, although ANNs are very popular, diverse, and powerful, they have essential disadvantages in applications related to the current study's domain, as previously mentioned. Thus, unlike the ANN approach used in the original study [40] to avoid the weaknesses of ANNs, the present research used Random Forest (RF) as the ML technique to induce the ML models.

RF is a decision tree-based ML technique using the ensemble method principle by averaging multiple DTs to improve predictive accuracy and control overfitting. This approach leverages the strengths of multiple DTs, each trained on random subsets of the data and features, to produce a more robust model than any single tree could offer. RF significantly reduces the variance without substantially increasing bias by aggregating the predictions from many trees through majority voting for classification tasks or averaging for regression tasks [72].

RFs have many advantages over ANNs. RF requires smaller datasets than ANNs to perform similarly, making the ML approach more suitable for situations with limited data availability [73]. It also requires shorter training times and less advanced hardware for training [74]. Unlike ANNs, RFs can handle categorical and numerical data without extensive preprocessing or feature scaling and require much simpler hyperparameter tuning, not requiring highly specialized staff to use them [73]. Although hyperparameter optimization has important implications for the performance of the ML model, since RF is less dependent on it and ML technique simplicity was a study requirement, the RF classifier default settings in the WEKA framework were used in all experiments in the present study. The underlying idea was that if the cost-sensitive approach using RF with default hyperparameter values could reach good results using data from multiple in-company projects with some distinct programming languages, that would be compelling enough since it could indicate that even superior results could be achieved with proper hyperparameter tuning. Additionally, that would demonstrate software testing professionals without much experience in ML would be able to use it in practical applications.

Moreover, RF is less prone to overfitting than ANNs because of the ensemble method of averaging multiple DTs, which leads to better generalization by reducing variance, which is essential in the investigated domain [72]. Finally, although RFs are not entirely white-box, they have higher explainability and interpretability than ANNs since their induced decision paths through the trees can be examined [75]. Feature importance scores can be generated, offering insights into model decisions, and supporting managers' decision-making on policies and actions to improve software quality in future development iterations. Thus, higher explainability and interpretability could augment the utility of the research domain in using ML to identify defective modules.

2.3. Experimental Protocol

The same protocol used in the original study [40] protocol was replicated here. However, a distinct ML classifier type was used in the present study to support its goals. Like in [40], a cost-sensitive approach was used in training to compensate for the harmful effect of the unbalanced dataset in generating the high FN ratios. Therefore, distinct cost values were assigned as a penalty to the FNs to reduce the ML model bias towards the most represented class (non-defective) without changing the cost of false positives (FP), which creates relative costs (RC) effect between the FN and FP penalty values. That aims to understand the effect of assigning different cost values to the quality of the final model predictions and lower the FN rate. However, a larger range of RC values assigned to FN ({1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 20, 30, 40, 50, 100}) was experimented to broaden the investigation of the effects of the cost-sensitive training in the present domain. Since, as in [76,77], distinct, and a larger number of ML classifiers (RF) models were induced when compared to [40]. The dataset composition and research protocol are depicted in Figure 2.

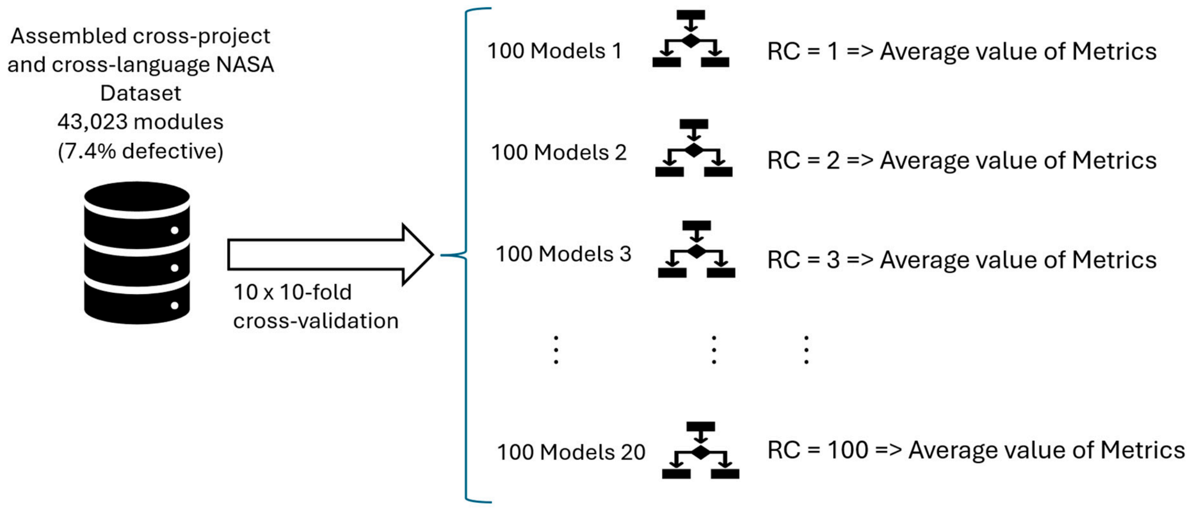

Assembled cross-project and cross-language NASA Dataset
43,023 modules
(7.4% defective)
10 x 10-fold cross-validation
100 Models 1
RC = 1 => Average value of Metrics
100 Models 2
RC = 2 => Average value of Metrics
100 Models 3
RC = 3 => Average value of Metrics
100 Models 20
RC = 100 => Average value of Metrics

Figure 2. Protocol used to create distinct models to predict defective software modules using different RC values to support decision-making.

As in the benchmark study [40], for each distinct cost value assigned for the FN, distinct ML classifiers (RF) were induced using a 10 × 10-fold cross-validation strategy [78]. The 10 × 10-fold cross-validation supports a more reliable validation of the proposed technique. Among the arguments by [79], the 10-fold was used rather than the leave-one-out cross-validation because it yields better results for the size of the dataset and results in less variance, which helps to compare the performance of distinct ML models induced as the FN assigned cost is increased. It smooths out the extreme effects of the luckiest and unluckiest data selection for training and testing, which leads to more realistic conclusions. Moreover, compared to vanilla train/test dataset split strategies, it reduces problems like underfitting and overfitting and helps to estimate better how accurately the model will perform in practice.

For all the executions, a unitary cost (1) was assigned to true positive (TP), true negative (TN), and FP. That is, the default penalty value (=1) was kept for TP, TN, and FP to train all the ML models in this study. Moreover, the default FN penalty value (=1) was also maintained for the experiments investigating RC = 1 (baseline). However, for all other RC values, the FN cost (penalty value) was set equal to the corresponding investigated RC value. Thus, in the setup for the experiment to investigate RC = 2, FN cost was set to 2. In the setup for the experiment to investigate RC = 3, FN cost was set to 3, and so on, until FN was set to 100 for RC = 100. Since 20 values of RC were investigated, 20 executions of 10 × 10-fold cross-validation were executed in total, as illustrated in Figure 2. Therefore, for each value of RC explored, 10 repetitions of 10-fold cross-validation were performed, resulting in 100 ML models trained and validated for each distinct cost assigned to FN. Finally, for each one of the 20 executions, the average value of the evaluation metrics (detailed in the next subsection) from the 100 RF was computed and further analyzed (as presented in Section 3). Since 20 distinct FN cost values were used, a total of 2000 RFs were generated considering all the executions.

Following the 10-fold cross-validation protocol, for each execution, the NASA MDP CROSS-PROJECTS DATASET was split into 10 equal-size subsets. Then, all 10 possible combinations of 9 subsets were used to train the RF, and each subset left aside per combination was used to evaluate the performance of the ML model. Since a 10 $\times$ 10-fold cross-validation was used, that process was repeated 10 times per value of RC evaluated. It is noteworthy that for each repetition, a random split of the dataset into 10 equal-size subsets was performed. Therefore, using this protocol, 100 random combinations of data seen (for ML training) and unseen (for ML performance evaluation) among the multiple projects inside an organization were obtained.

2.4. Evaluation Metrics

Various metrics were collected or computed to evaluate the ML model's performance. They were all average values computed from the 100 samples measured from the RFs induced for each cost value assigned. The fundamental metrics collected were those from the confusion matrix. The **true positive (TP)** is the number of defective software modules correctly classified as defective by the ML model. Thus, they correctly inform the software testing team about the modules that must be considered in the software testing scope because they are defective, using the available resources appropriately. The **true negative (TN)** is the number of non-defective software modules correctly classified as non-defective by the ML model. Thus, they correctly inform the software testing team about the modules that could be left outside the software testing scope since they are not defective, saving the available resources appropriately. The **false positive (FP)** is the number of non-defective software modules incorrectly classified as defective by the ML model. Thus, they wrongly induce the software testing team to consider them inside the testing scope, although they are not defective, wasting resources, which reduces their efficiency. The **false negative (FN)** is the number of defective software modules incorrectly classified as non-defective by the ML model. Thus, they wrongly induce the software testing team to leave those defective modules outside the testing scope, reducing their efficacy. Therefore, FNs are dangerous and must be avoided since those defective modules can cause severe consequences when the software operates in production.

When managers design the software testing scope informed by the ML classifier, they include all the modules classified as defective (TP + FP). Thus, the metric **number of modules tested (MT)** is defined by Equation (1) [40],

$$MT = TP + FP. \tag{1}$$

Therefore, using a decision-making process informed by the ML classifier, managers will exclude from the software testing scope the modules indicated as non-defective (TN + FN). Thus, Equation (2) defines the metric **number of modules not tested (MNT)**,

$$MNT = TN + FN. \tag{2}$$

When the ML model supports decision-making, the result is a reduction in software testing scope, according to [40]. Equation (3) defines the metric **scope reduction (SR)** [40],

$$SR = \frac{MNT}{MT + MNT} \tag{3}$$

On the other hand, the fraction of the total number of modules suggested by the ML classifier as the proper software testing scope is the **relative test scope (RTS)** [40], defined in Equation (4),

$$RTS = 1 - SR. \tag{4}$$

Cost-sensitive training is influenced by the relationship between the costs assigned to FN (C_{FN}) and FP (C_{FP}). Therefore, the approach core strategy is increasing the **relative cost (RC)** [40], defined by Equation (5), between the cost assigned to FN and FP and evaluating the average performance of the ML models.

$$RC = \frac{C_{FN}}{C_{FP}}, \tag{5}$$

where, in the present research, C_{FP} = 1, thus,

$$RC = C_{FN}. \tag{6}$$

Since $C_{FN} \subset \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 20, 30, 40, 50, 100\}$, RC $\subset \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 20, 30, 40, 50, 100\}$.

As defined by [40], **Precision (P)** [80], in this research domain, translates the **efficiency (Eff)** of the test effort, because it represents the total number of defective modules detected from the total number of modules tested. Ideally, software testing effort should be spent only on defective modules indicated by a 100% efficient ML model. Equation (7) indicates the expression used to compute the Model's Eff [40],

$$Eff = P = \frac{TP}{(TP + FP)}. \tag{7}$$

Analogously, according to [40], **Recall (R)** [80] can be referred to as **efficacy (Ef)**, since it indicates how effective the test effort can be following, considering exactly the software testing scope suggested by the ML model. Since the software testing goal is to discover 100% of the defective modules in the system, R measures the fraction of the goal achieved by the test effort informed by the ML model. A software testing scope delineated by a 100% effective ML model would discover all the defective modules. Equation (8) indicates the expression used to compute the Model's Ef [40].

$$Ef = R = \frac{TP}{(TP + FN)}. \tag{8}$$

Furthermore, the ML model **Accuracy (Acc)** [80,81] indicates the ratio of software modules correctly classified (TP + TN) from the total number of modules (TP + TN + FP + FN). A 100% accurate ML model would result in no misclassification, that is, nor FP or FN. Although that seems highly desirable, paradoxically, an ML model with 100% accuracy usually has an overfit, indicating compromised generalizability. That is highly undesirable since it reduces its practical application. Equation (9) indicates the expression used to compute the Model's Acc [80,81].

$$ACC_{RC} = \frac{(TP_{RC} + TN_{RC})}{(TP_{RC} + TN_{RC} + FP_{RC} + FN_{RC})}. \tag{9}$$

As used by [40], a benchmark based on the unitary cost ML Model was used to evaluate how software testing efforts using the scope suggested by the induced ML models with higher RC values performed (Eff and Ef) compared to those with $RC = 1$. Thus, the **relative efficiency to the unitary cost ML model (REffU)** and the **relative efficacy to the unitary cost ML model (REfU)** were computed for each RC > 1 to support those comparisons using Equations (10) and (11), respectively.

$$REffU_{RC} = \frac{Eff_{RC}}{Eff_{RC=1}}, \tag{10}$$

$$REfU_{RC} = \frac{Ef_{RC}}{Ef_{RC=1}}. \tag{11}$$

As suggested by [40], another benchmark was used to evaluate how software testing efforts using the scope suggested by the induced ML models performed (Eff and Ef) compared to similar software testing efforts with identical scope sizes but based on the random selection of modules, representing a decision-making not informed by the ML models. The **efficiency of the random selection (EffR)** was computed using Equation (12), which $\boldsymbol{p_{TP}}$ is the **probability of a defective module being selected randomly**, which is 7.4% for the assembled dataset used in this study and is not affected by RC values. The **efficacy of the random selection (EfR)** was computed for each RC to support those comparisons using Equation (13).

$$EffR_{RC} = p_{TP} = 7.4\%, \tag{12}$$

$$EfR_{RC} = EffR_{RC} \times RTS_{RC} = 7.4\% \times RTS_{RC}. \tag{13}$$

Aiming to compare Eff to EffR and Ef to EfR, some additional ratios were computed. The **relative efficiency to the random selection (REffR)** was computed using Equation (14). The **relative efficacy to the random selection (REfR)** was computed for each RC to support those comparisons using Equation (15).

$$REffR_{RC} = \frac{Eff_{RC}}{EffR_{RC}}, \tag{14}$$

$$REfR_{RC} = \frac{Ef_{RC}}{EfR_{RC}}. \tag{15}$$

As in the original study [40], other performance comparisons were performed using the metric **Relative Percent Correct (RPC)**, which represents the ratio of the number of modules classified correctly by the ML model to the number of modules classified correctly by each benchmark. The **Relative Percent Correct relative to the Unitary cost ML model benchmark (RPCU)** was computed using Equation (16), while the **Relative Percent Correct relative to the Random selection of modules (RPCR)** was computed using Equation (17).

$$RPCU_{RC} = \frac{ACC_{RC}}{ACC_{RC=1}}, \tag{16}$$

$$RPCR_{RC} = \frac{ACC_{RC}}{7.4\%}. \tag{17}$$

The metric **Misclassified Defective Modules (MDM)** indicates the ratio of the number of defective modules misclassified as non-defective by the ML model to the total number of existing defective modules in the system (3196 in the assembled dataset, and k is the number of folds). The metric **Misclassified Non-defective Modules (MNDM)** indicates the ratio of the number of non-defective modules misclassified as defective by the ML model to the total number of existing non-defective modules in the system (39827 in the assembled dataset, and k is the number of folds). Those metrics were computed for each RC value using Equations (18) and (19), respectively.

$$MDM_{RC} = \frac{FN_{RC}}{\frac{3196}{k}}, \tag{18}$$

$$MNDM_{RC} = \frac{FP_{RC}}{\frac{39827}{k}}. \tag{19}$$

Finally, the metric **Unnecessary Tests (UT)** [40] was computed to evaluate the ratio of module tests that were wasted because they were unnecessary. Equation (20) shows how the UT was calculated for each value of RC evaluated,

$$UT = \frac{FP_{RC}}{MT_{RC}}. \tag{20}$$

2.5. Statistical Tests

A paired *t*-test (with correction) was used to evaluate if the average value of the evaluated metrics for each RC greater than 1 was statistically significant at a 5% level (p-value ≤ 0.05) compared to the average value of the equivalent metric for RC = 1 (benchmark with no increment of penalty value for FN). Thus, in all the results, the metrics values found to have statistically significant differences compared to their benchmarks (RC = 1) were marked with "*".

3. Results

The protocol described in the previous section was executed entirely, providing a dataset of results with the metrics used to support the analysis presented here. In all tables with results, "*" indicates the statistical significance at a 5% level of the paired *t*-test with correction (compared to the benchmark, $RC = 1$).

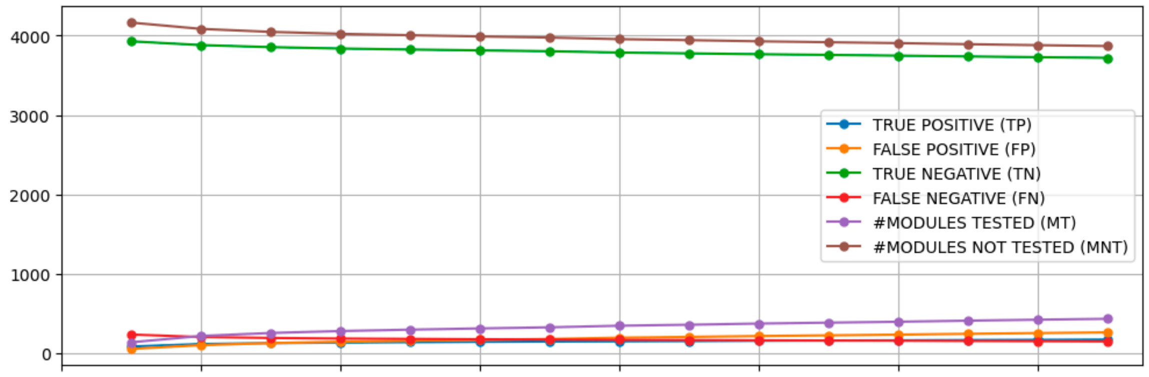
Table 2 shows the models' accuracies for each relative cost. A paired *t*-test (with correction) was used to compare the accuracies' averages with the significance test performed at a 5% level. The statistical significance of the *t*-test is indicated by "*" where the p-value was lower than 5% (compared to the benchmark, $RC = 1$). Like in the reference study [40], as RC increases, the accuracy decreases. However, here, the reduction was only 2.2% (93.27% to 91.19%), much smaller than that observed in the reference study, where it was reduced to almost half (52%) when the RC was 10 times higher. Notably, an average accuracy of 92.27% cannot be considered a good result for a model trained with an imbalanced binary dataset, with 92.6% of the instances belonging to the most representative class (non-defective module). Finally, all values of ACC (RC > 1) were statistically significant compared to its baseline value (RC = 1) at a 5% level.

Table 2. TP, TN, FP, FN, MT, MNT, SR × RC.

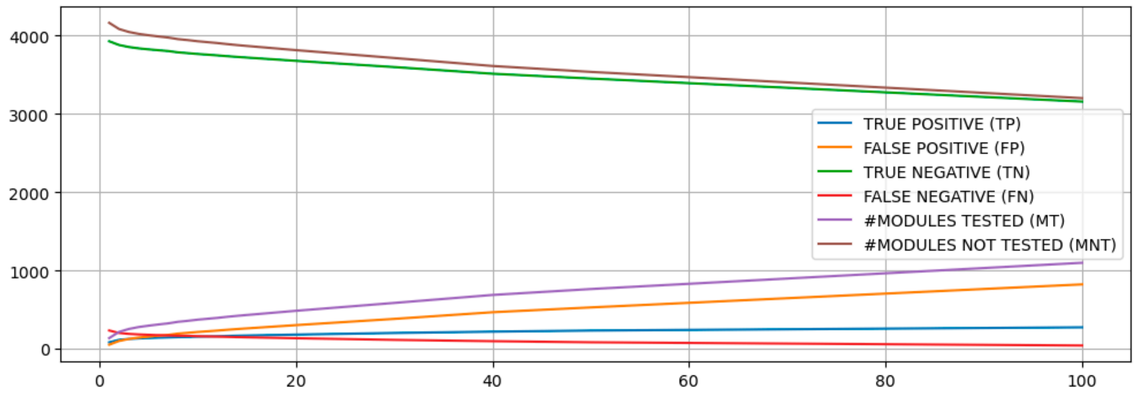
RC	ACC	TP	FP	TN	FN	MT	MNT	SR
1	93.3%	84.04	54.11	3928.59	235.56	138.15	4164.15	96.8%
2	92.9% *	115.65 *	101.62 *	3881.08 *	203.95 *	217.27 *	4085.03 *	94.9% *
3	92.6% *	127.42 *	127.5 *	3855.2 *	192.18 *	254.92 *	4047.38 *	94.1% *
4	92.3% *	134.5 *	144.87 *	3837.83 *	185.1 *	279.37 *	4022.93 *	93.5% *
5	92.2% *	139.28 *	157.48 *	3825.22 *	180.32 *	296.76 *	4005.54 *	93.1% *
6	92.0% *	143.87 *	168.35 *	3814.35 *	175.73 *	312.22 *	3990.08 *	92.7% *
7	91.8% *	147.32 *	179.02 *	3803.68 *	172.28 *	326.34 *	3975.96 *	92.4% *
8	91.5% *	150.83 *	195.57 *	3787.13 *	168.77 *	346.4 *	3955.9 *	91.9% *
9	91.3% *	152.98 *	205.86 *	3776.84 *	166.62 *	358.84 *	3943.46 *	91.7% *
10	91.2% *	157.08 *	216.51 *	3766.19 *	162.52 *	373.59 *	3928.71 *	91.3% *
11	91.1% *	159.89 *	225.09 *	3757.61 *	159.71 *	384.98 *	3917.32 *	91.1% *
12	90.9% *	162.06 *	234.29 *	3748.41 *	157.54 *	396.35 *	3905.95 *	90.8% *
13	90.7% *	165.53 *	244.49 *	3738.21 *	154.07 *	410.02 *	3892.28 *	90.5% *
14	90.6% *	168.42 *	253.82 *	3728.88 *	151.18 *	422.24 *	3880.06 *	90.2% *
15	90.5% *	171.04 *	262.29 *	3720.41 *	148.56 *	433.33 *	3868.97 *	89.9% *
20	89.8% *	183.01 *	303.94 *	3678.76 *	136.59 *	486.95 *	3815.35 *	88.7% *
30	88.4% *	203.88 *	383.56 *	3599.14 *	115.72 *	587.44 *	3714.86 *	86.3% *
40	86.8% *	220.73 *	468.54 *	3514.16 *	98.87 *	689.27 *	3613.03 *	84.0% *
50	85.7% *	234.74 *	529.99 *	3452.71 *	84.86 *	764.73 *	3537.57 *	82.2% *
100	79.8% *	275.76 *	824.1 *	3158.6 *	43.84 *	1099.86 *	3202.44 *	74.4% *

Note: A paired *t*-test (with correction) was used to compare the accuracies' averages at a 5% significance level. "*" indicates p-values lower than 5% (compared to the benchmark, RC = 1).

Table 2 also shows the information from the confusion matrix (TP, TN, FP, FN), indicating how its distribution and RC change. All values of TP, FP, TN, FN, and MT for RC > 1 were statistically significant compared to their baseline values (RC = 1) at a 5% level. As found in the reference study, increasing RC results in increasing TP and decreasing FN, which is positive for using ML models to support test effort allocation. However, while the TP almost doubled from $RC = 1$ to 10, it increased to over seven times in the reference study using a different classifier on a single project and language dataset. Naturally, the increase in TP and the decrease in FN could only happen with an increase in FP (~4×) and a slight decrease in TN (4%). Consequently, as shown in Table 2, the number the classifier indicates to be tested (MT) grows as RC increases. That growth (2.7×) is lower than observed in the reference study (18.4). Moreover, while SR was reduced from 96% to 29% in the benchmark research, the reduction was much more moderate (96.8% to 91.3%) for the protocol run. Figure 3 illustrates the behavior of TP, TN, FP, FN, MT, and MNT over the distinct costs. Like all charts with results in the present study, the lower chart represents the full range of RC evaluated, and the upper one zooms in on RC [1,15] to better observe the initial behavior in a range compatible with the benchmark study. It is worth noting that the findings reveal a pattern of decreasing marginal returns leading to a saturation of earnings with the technique, demonstrating an asymptotic behavior that persists from the initial points onward. Finally, all values of MT, MNT, and SR (RC > 1) were statistically significant compared to their baseline values (RC = 1) at a 5% level.



(**a**) Initial RC range



(**b**) Full RC range

Figure 3. {TP, TN, FP, FN, MT and MNT} × RC.

Table 3 shows the classifiers' test efficiency and efficacy metrics and a theoretical benchmark obtained with the expected results from a random selection of modules to be tested with the same scope size for each RC. Except for EffR, all other metrics in Table 3 were statistically significant for RC > 1 compared to their baseline values (RC = 1) at a 5% level. The TP, TN, FP, and FN changes have essential implications for the efficiency, efficacy, and scope of software testing activities. The RC increase implicated in ML models resulted in lower test efficiency and higher efficacy with an increase of RTS, which corroborated the benchmark study's findings. However, in the benchmark study [40], the efficiency is reduced to 41.4% when $RC = 10$, while in the present study, a much lower reduction for the same RC was observed, equivalent to 68.9% of the initial one, indicating a smoother effect on the efficiency. In the same way, the efficacy increased by 7.6×, comparing the model with $RC = 10$ to $RC = 1$ in the benchmark study [40], while in the present study, a smoother effect was observed since it was increased by 1.9×. Finally, a smoother effect was also observed for RTS (an increase of 2.7×) compared to the reference study (an increase of 17.8×).

Table 3. EFFICIENCY and EFFICACY (Models and Random Benchmarks), RTS, RELATIVE EFFICIENCY, EFFICACY and RPC (Benchmark: Unitary Cost ML Model and Benchmark: Random Selection) × RC.

RC	**ML Model**			**Benchmarks**							
				Random Selection					**Unitary Cost ML Model**		
	Eff	**Ef**	**RTS**	**EffR**	**EfR**	**REffR**	**REfR**	**RPC**	**REffU**	**REfU**	**RPC**
1	61.0%	26.0%	3.2%	7.4%	0.2%	821.2%	10,899.8%	1255.6%	100.0%	100.0%	100.0%
2	53.0% *	36.0% *	5.1% *	7.4%	0.4% *	713.5% *	9596.2% *	1250.6% *	86.9% *	138.5% *	99.6% *
3	50.0% *	40.0% *	5.9% *	7.4%	0.4% *	673.1% *	9087.6% *	1246.1% *	82.0% *	153.8% *	99.2% *
4	48.0% *	42.0% *	6.5% *	7.4%	0.5% *	646.2% *	8706.9% *	1242.9% *	78.7% *	161.5% *	99.0% *
5	47.0% *	44.0% *	6.9% *	7.4%	0.5% *	632.7% *	8587.0% *	1240.5% *	77.0% *	169.2% *	98.8% *
6	46.0% *	45.0% *	7.3% *	7.4%	0.5% *	619.2% *	8347.3% *	1238.5% *	75.4% *	173.1% *	98.6% *
7	45.0% *	46.0% *	7.6% *	7.4%	0.6% *	605.8% *	8163.6% *	1236.2% *	73.8% *	176.9% *	98.5% *
8	44.0% *	47.0% *	8.1% *	7.4%	0.6% *	592.3% *	7858.0% *	1232.1% *	72.1% *	180.8% *	98.1% *
9	43.0% *	48.0% *	8.3% *	7.4%	0.6% *	578.8% *	7747.0% *	1229.6% *	70.5% *	184.6% *	97.9% *
10	42.0% *	49.0% *	8.7% *	7.4%	0.6% *	565.4% *	7596.2% *	1227.6% *	68.9% *	188.5% *	97.8% *
11	42.0% *	50.0% *	8.9% *	7.4%	0.7% *	565.4% *	7521.9% *	1225.8% *	68.9% *	192.3% *	97.6% *
12	41.0% *	51.0% *	9.2% *	7.4%	0.7% *	551.9% *	7452.2% *	1223.5% *	67.2% *	196.2% *	97.4% *
13	40.0% *	52.0% *	9.5% *	7.4%	0.7% *	538.5% *	7345.0% *	1221.5% *	65.6% *	200.0% *	97.3% *
14	40.0% *	53.0% *	9.8% *	7.4%	0.7% *	538.5% *	7269.6% *	1219.5% *	65.6% *	203.8% *	97.1% *
15	39.0% *	54.0% *	10.1% *	7.4%	0.7% *	525.0% *	7217.2% *	1217.6% *	63.9% *	207.7% *	97.0% *
20	38.0% *	57.0% *	11.3% *	7.4%	0.8% *	511.5% *	6779.3% *	1208.3% *	62.3% *	219.2% *	96.2% *
30	35.0% *	64.0% *	13.7% *	7.4%	1.0% *	471.2% *	6309.7% *	1190.0% *	57.4% *	246.2% *	94.8% *
40	32.0% *	69.0% *	16.0% *	7.4%	1.2% *	430.8% *	5797.7% *	1168.6% *	52.5% *	265.4% *	93.1% *
50	31.0% *	73.0% *	17.8% *	7.4%	1.3% *	417.3% *	5528.5% *	1153.8% *	50.8% *	280.8% *	91.9% *
100	25.0% *	86.0% *	25.6% *	7.4%	1.9% *	336.5% *	4528.5% *	1074.6% *	41.0% *	330.8% *	85.6% *

Note: A paired *t*-test (with correction) was used to compare the accuracies' averages at a 5% significance level. "*" indicates *p*-values lower than 5% (compared to the benchmark, RC = 1).

Furthermore, Table 3 and Figure 4 support a comparison between the software testing efforts informed by ML models and the benchmark's performance based on a non-informed approach, where the modules for software testing are selected randomly. Notably, in the present study, the random benchmarks are worse than in the reference study because of the higher imbalance of the dataset used here. In the original study, the ratio of defective classes is 2.5, the number of defective classes in the current dataset, resulting in a random benchmark efficiency 2.5× higher than the one obtained here. The results indicate that, despite the RC value, the informed approach outperforms the non-informed approach, demonstrating superior performance. That corroborates findings from the benchmark study. Moreover, it also aligns with the field literature since, despite the existing gaps and limitations, the results indicate that even a suboptimal ML model can improve the software testing performance and outperform a non-informed approach.

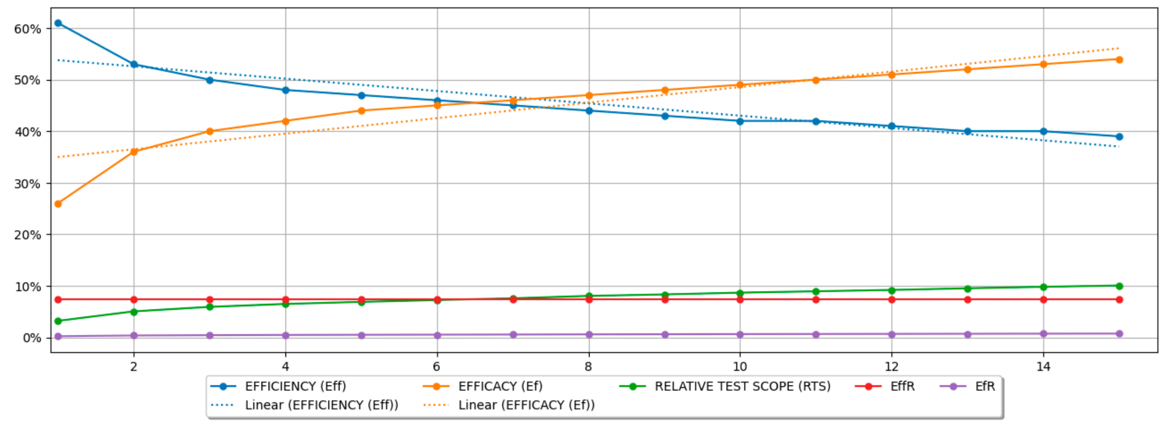
Aiming to compare, quantitatively, the ratio of the decrease in efficiency with the increase in efficacy as RC is increased, a linear model was built with a linear regression to explain the efficacy and efficiency behaviors, having RC as the independent variable, resulting in Equations (21) and (22). The coefficient of determination (R^2) was computed to measure the proportion of variance in the dependent variables that is predictable from the independent variable (RC) for each regression model. For any regression, R^2 is calculated as

$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$, where SS_{res} (Residual Sum of Squares) is the sum of the squared differences between the observed values and the values predicted by the model, and SS_{tot} (Total Sum of Squares) is the sum of the squared differences between the observed values and the mean of the observed values.

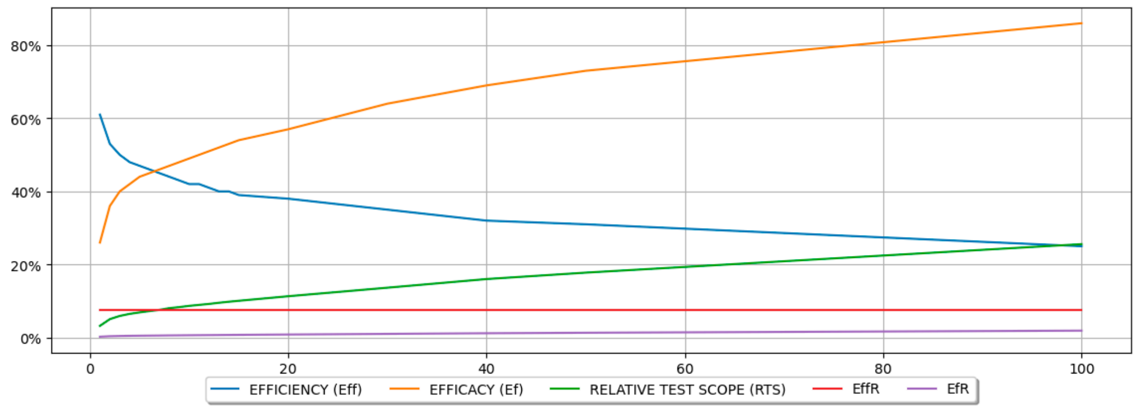
$$Eff = (0.0151 \cdot RC + 0.3348) \times 100\%;\ R^2 = 0.8394, \tag{21}$$

$$Ef = (-0.012 \cdot RC + 0.5497) \times 100\%;\ R^2 = 0.8322. \tag{22}$$

It is important to highlight that all linear regressions conducted in this study were tailored to RC ranges from 1 to 15 for two primary reasons. Firstly, the baseline study [40] had a narrower range of 1 to 10, necessitating an adjustment of parameters to facilitate a more accurate comparison. Secondly, the focal point of our current research lies in assessing the aggressiveness of initial RC range gains. Therefore, the extrapolation employed in these current experiments aimed to provide a comprehensive view of the models' performance in higher-cost scenarios.



(**a**) Initial RC range



(**b**) Full RC range

Figure 4. {Eff, Ef, RTS, EffR, and EfR} × RC.

The plots and descriptions of those models are illustrated in Figure 4. Since both models reached $R^2 > 80\%$, they can be considered suitable for explaining efficiency and efficacy variances by RC changes. Comparing the effect sizes of RC on efficiency (−0.0120) and efficacy (0.0151), each decrease in test efficiency (caused by the increase of the RC) results in an average rise of test efficacy that is 25.8% higher than the efficiency decrease. Thus, each RC unit increment returns an improvement in test efficacy higher by almost 26% (on average) than the price paid in test efficacy decrease. Since the reduction in efficiency is less than the improvement in efficacy, the same advantage observed in the benchmark study was demonstrated in the present study.

Those results indicate that conveniently adjusting RC makes finding an optimal equilibrium between efficiency and efficacy and the extent of testing coverage possible. This is the core idea of the approach, which has been demonstrated only for a single project dataset until now. Thus, it indicates that the approach's core idea can be generalized for a larger dataset encompassing multiple projects developed in distinct moments by different teams involving distinct technologies (programming languages).

Consequently, by adjusting the RC, test managers can use ML models to optimize the test scope according to the resources available for the software testing effort. Using ML models with lower RC will help prioritize a narrower scope of testing while maintaining high efficiency, which is advisable in scenarios where testing resources are constrained. On the other hand, in scenarios where available resources are less constrained, using higher RC values will help to expand the testing scope wisely, aiming for an improvement in efficacy despite a potential reduction in efficiency.

Additionally, Table 3 presents comparisons of efficiency and efficacy with other benchmarks. One of the benchmarks used was the efficiency and efficacy of the unitary cost model ($RC = 1$). Thus, for each RC value, the table shows how the ML model's performance (efficiency and efficacy) compares to the baseline model's performance ($RC = 1$). Table 3 also shows the RPC, supporting a comparison between the number of modules correctly classified by each model obtained for RC > 1 and the baseline ($RC = 1$). Chart (a) in Figure 5 shows the plot of the relative efficiency and efficacy, as well as the RPC, considering the unitary cost as a baseline.

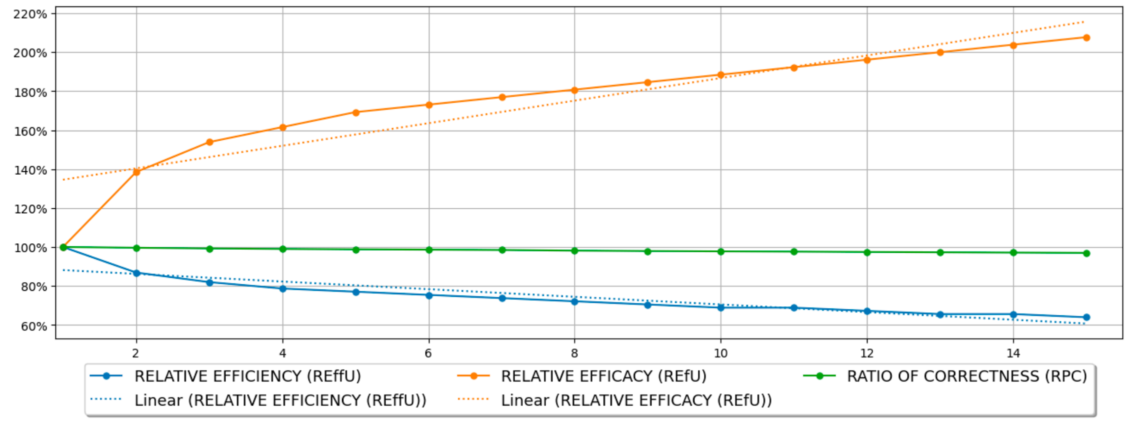
The same analysis performed for the absolute values of efficiency and efficacy was performed here to compare quantitatively the ratio of the decrease in relative efficiency with the increase in efficacy as RC is increased using a linear model. The expressions of regressions are shown in Equations (23) and (24),

$$REffU = (0.058 \cdot RC + 1.2875) \times 100\%; R^2 = 0.8394, \tag{23}$$

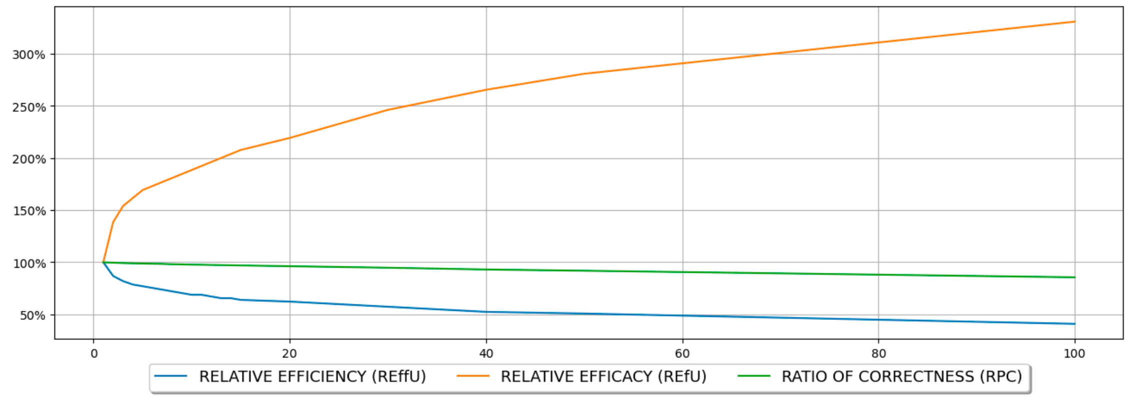
$$REfU = (-0.0196 \cdot RC + 0.9012) \times 100\%; R^2 = 0.8322. \tag{24}$$

The plots and descriptions of those models are illustrated in Figure 5. Since all the regressions reached $R^2 > 80\%$, they can be considered suitable for explaining relative efficiency and relative efficacy variances by RC changes. Comparing the effect sizes of RC on efficiency (−0.0196) and efficacy (0.0580), each decrease in the relative test efficiency (caused by the increase of the RC) results in an average rise of relative test efficacy that is 3× higher than the efficiency decrease. Thus, each RC unit increment returns an improvement in relative test efficacy 3× higher (on average) than the decline observed in relative test efficacy on average, corroborating the benchmark study's finding.

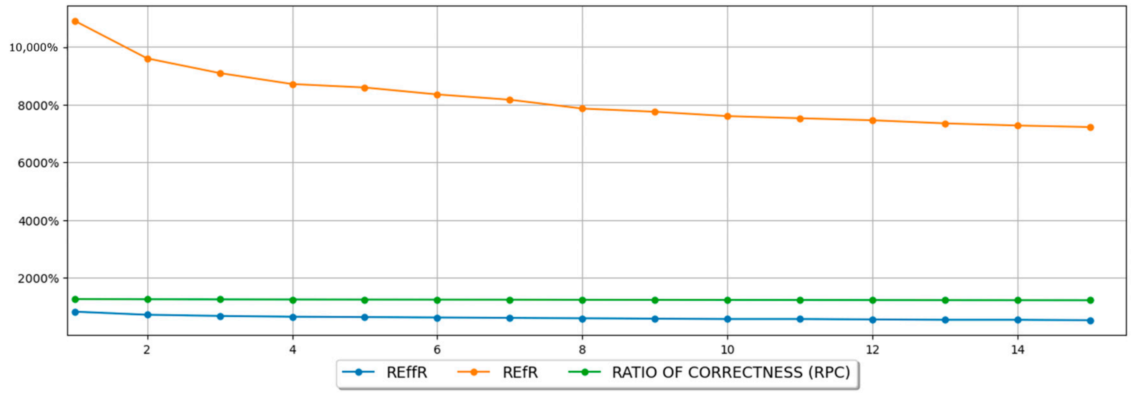
Another benchmark comparison shown in Table 3 was related to using the random benchmark. The test efficiency and efficacy reached by each ML model induced with distinct RC values were compared against the baseline value of non-informed software testing based on random module selection. Chart (c) in Figure 5 shows the plot of the relative efficiency, efficacy, and RPC, with the random benchmark as the baseline. The results also corroborate the benchmark study's findings, where the relative efficacy drops faster than the relative efficiency as RC is increased. Still, those values are always higher than 100%, demonstrating that the ML-based approach outperforms the non-informed selection of modules for testing. However, since an increase in RC implicates an increase in test scope, it is natural to expect that as the scope increases, it weakens the ML-based approach advantages since, ultimately, when 100% of the modules are tested, an ML-based approach offers no additional value when compared to the random selection of modules to be tested. Finally, charts (b) and (d) of Figure 5 show the same analyses for a more extensive RC range encompassing higher values.



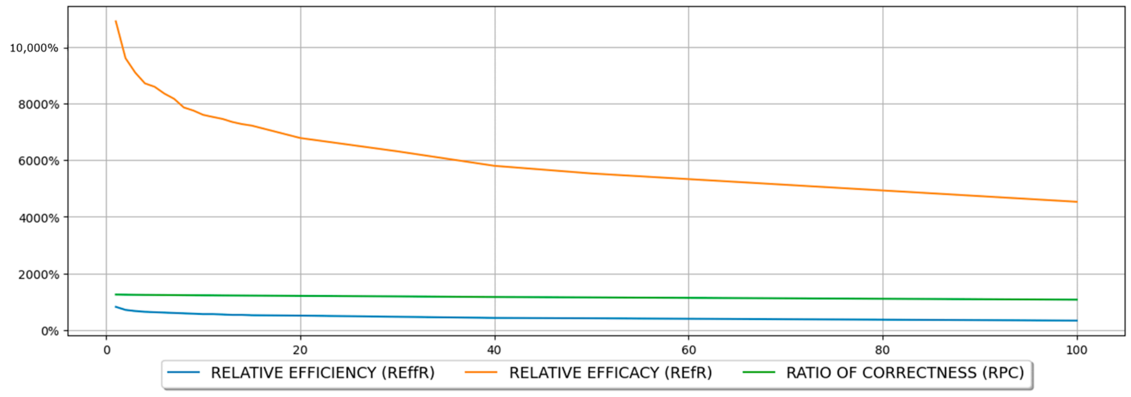
(**a**) Benchmark: Unitary Cost ML Model–Partial RC range.



(**b**) Benchmark: Unitary Cost ML Model–Full RC range.



(**c**) Benchmark: Random Selection–Partial RC range.



(**d**) Benchmark: Random Selection–Full RC range.

Figure 5. {REffR, RefR, RPC} × RC.

The last analysis was to understand how RC increases affect MDM, MNDM, and UT. Table 4 shows the values of each of those metrics for each RC value. All those metrics were statistically significant for RC > 1 compared to their baseline values (RC = 1) at a 5% level. The experiments were conducted using a 10-fold cross-validation approach. Thus, each fold contained, on average, 320 defective modules (used for MDM calculation) and 3983 non-defective modules (used for MNDM calculation) with $k = 10$. Figure 6 shows those metrics plotted for each RC value. The upper chart of Figure 6 is focused on a narrow range of RC, while the bottom one shows a chart encompassing the full range of RC.

The behavior of MDM, MNDM, and UT versus RC was compared using linear models obtained with linear regressions. The regressions expressions are shown in Equations (25)–(27),

$$UT = (0.0119 \cdot RC + 0.4509) \times 100\%; R^2 = 0.8284, \tag{25}$$

$$MDM = (-0.0151 \cdot RC + 0.6652) \times 100\%; R^2 = 0.8394, \tag{26}$$

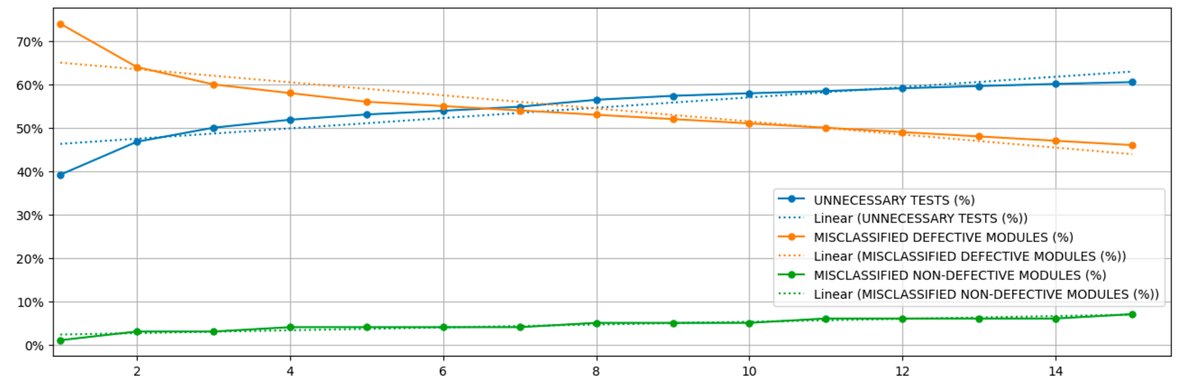
$$MNDM = (0.0033 \cdot RC + 0.0197) \times 100\%; R^2 = 0.8997. \tag{27}$$

Those models reached values R^2 greater than 80%, indicating they could adequately explain the variance of those metrics using the independent variable RC. The linear model expressions are expressed in the chart. The MDM, MNDM, and UT effect sizes in the models were -0.0151, 0.0033, and 0.0119, respectively. Those coefficients indicate that each increase in RC causes a reduction in MDM that is 26.9% higher than the increase it causes in UT. Moreover, they suggest that each rise in RC causes a decrease of 457.6% in MDM, which is higher than the increase it causes in MNDM. MDMs are dangerous, especially in safety-critical systems, because they divert the software testing effort to evaluate properly those misclassified defective classes, increasing the risks of failure during customer use. Thus, they are highly undesired. Using the proposed approach, the MDM is reduced in a ratio much higher than it increases the MNDM. Although MNDMs are undesired, their negative outcome is to induce the software testing effort in testing a non-defective software module, which wastes resources. Still, they do not cause dangerous situations that could cause more severe losses, such as jeopardizing life or property.

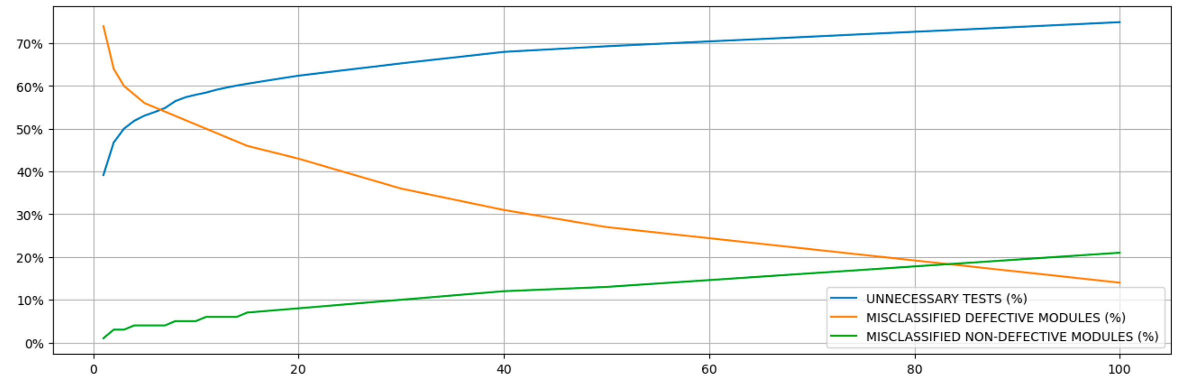
Table 4. MDM, MNDM, and UT × RC.

RC	MDM	MNDM	UT
1	74.0%	1.0%	39.2%
2	64.0% *	3.0% *	46.8% *
3	60.0% *	3.0% *	50.0% *
4	58.0% *	4.0% *	51.9% *
5	56.0% *	4.0% *	53.1% *
6	55.0% *	4.0% *	53.9% *
7	54.0% *	4.0% *	54.9% *
8	53.0% *	5.0% *	56.5% *
9	52.0% *	5.0% *	57.4% *
10	51.0% *	5.0% *	58.0% *
11	50.0% *	6.0% *	58.5% *
12	49.0% *	6.0% *	59.1% *
13	48.0% *	6.0% *	59.6% *
14	47.0% *	6.0% *	60.1% *
15	46.0% *	7.0% *	60.5% *
20	43.0% *	8.0% *	62.4% *
30	36.0% *	10.0% *	65.3% *
40	31.0% *	12.0% *	68.0% *
50	27.0% *	13.0% *	69.3% *
100	14.0% *	21.0% *	74.9% *

Note: A paired t-test (with correction) was used to compare the accuracies' averages at a 5% significance level. "*" indicates p-values lower than 5% (compared to the benchmark, RC = 1).



(**a**) Initial RC range



(**b**) Full RC range

Figure 6. {MDM, MNDM, and UT} × RC.

The consistent statistically significant differences obtained for RC > 1 for all the evaluated metrics compared to their baseline values reinforce the promising results of the cost-sensitive approach found in [40] and confirm them in the specific in-company cross-project and cross-language scenario evaluated in the present study. The results show that the approach can be very appealing in terms of informing the software testing plan since it can improve the quality of the software testing effort, making it possible to accomplish more with the same (or less) available resources. Therefore, besides corroborating the benchmark study's findings, this last analysis demonstrated that the cost-sensitive approach can potentially suit hybrid software development environments involving diverse projects with distinct programming languages and software development teams.

4. Discussion

This section discusses the implications of software testing scope decision-making based on the cross-project and cross-language ML models induced using the cost-sensitive approach in a hypothetical scenario.

The test manager has a budget of n_t software testers and n_t days available for software testing. Equation (28) gives the **available software testing budget in hours (B)**, considering a workday of 8 h/day,

$$B = 8 \times n_t \times n_d \tag{28}$$

In a comparative hypothetical situation, as presented in Figure 7, 8 software testers are available for a 17-day effort, resulting in B equal to 1088 h. The project's average effort to test a software module (E) was estimated to be 5 h. Thus, the available budget can afford to test around 218 modules on average. In Table 2, the closest MT value to 218 is 217, which corresponds to $RC = 2$; therefore, the ML model to be used to support the decision-making on software testing must be one that is trained with $RC = 2$.

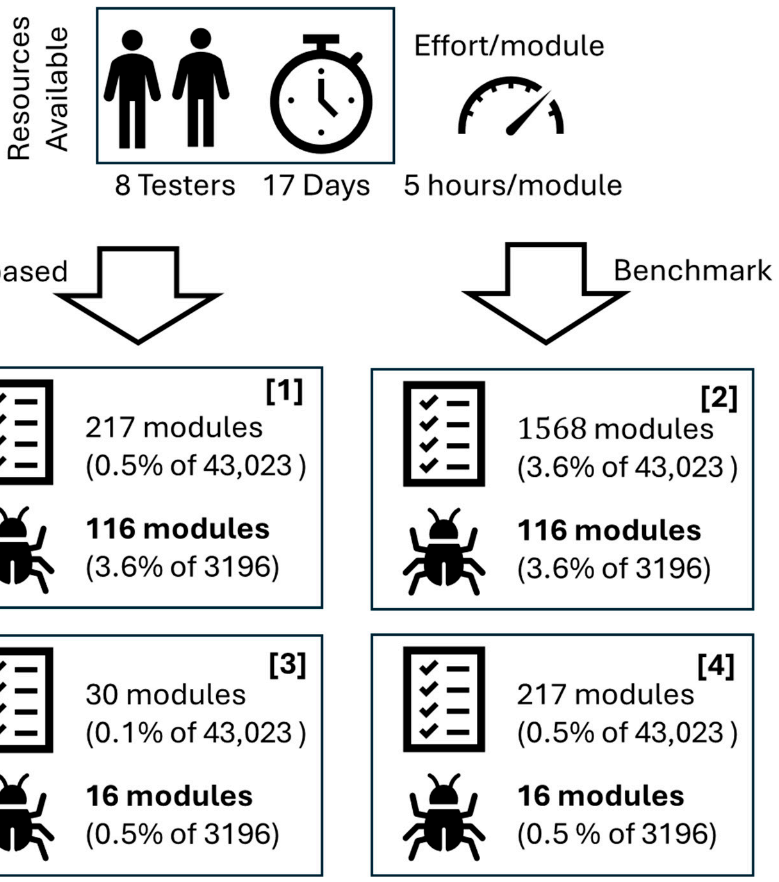


Figure 7. Comparison of the value added by the proposed technique in a hypothetical scenario with four decisions regarding the software testing scoping and allocation of available software testing resources.

When $RC = 2$, the average number of defective modules discovered by a software testing effort following the ML model corresponding scope is 116 (Figure 7([1])), according to Table 2. If the same software testing effort was performed in a same-size scope encompassing modules randomly selected, only 16 defective modules would be discovered (Figure 7([3])). That **equivalent number of defective modules discovered with random selection (DR)** is defined by Equation (29).

$$DR = p_{TP} \times \text{MT} = 7.4\% \times 217 = 16 \tag{29}$$

Thus, using the same budget available, the effort would identify over seven times more defective modules when an ML model-based decision-making process is used rather than a random selection of modules, which is quite impressive (Figure 7([1]) × Figure 7([3])). Moreover, a manager would need a **testing effort (TE_{Random})** of 7840 h (given by Equation (30)) to test 1568 software modules randomly picked to accomplish an equivalent performance in the number of defective modules discovered (Figure 7([1]) × Figure 7([2])). In other words, the manager would need a software testing budget 7.2 times higher without using the ML model to select the modules to be tested for achieving the same accomplishments, which is quite impressive.

$$TE_{Random} = \frac{TP}{p_{TP}} \times E = \frac{116}{7.4\%} \times 5 = 7840. \tag{30}$$

Considering this hypothetical scenario, following the ML model recommendations, the software testing **productivity (Prod)** would be 9.4 h to find each defective module on average, given by Equation (31).

$$Prod = \frac{MT}{TP} \times E = \frac{217}{116} \times 5 = 9.4. \tag{31}$$

Thus, to reach the **equivalent result of the random selection (ER)** (16 defective modules), the software testing team informed by the ML model would need to test around 30 modules (Figure 7([3]) × Figure 7([4])), according to Equation (32), which would require a total effort of only 30 × 9.4 = 282 h,

$$ER = \frac{MT}{TP} \times DR = \frac{217}{116} \times 16 \approx 30 \tag{32}$$

Those results demonstrate the power of using ML models to support decision-making on software testing scope definition and resource allocation, helping quality assurance efforts accomplish better results with the available resources or even using fewer resources.

5. Conclusions

The current research validated the generalizability of the original study's findings in a broader scenario. While the original study used MLP classifiers and a single project dataset based on a single programming language, the present research demonstrated the generalizability of the cost-sensitive approach across multiple projects with multiple languages within the same organization. The NASA MDP datasets were used because they belong to the same organization and involve various languages. Although it was in a specific scenario, expanding the generalizability comprehension of the cost-sensitive approach in this domain is one of the present research's relevant contributions since it demonstrates that the approach can potentially be useful and reliable across various situations involving distinct projects, languages, and teams inside the same organization, indicating that it can be effective in practical applications. However, more investigation into broader generalization scenarios is needed, such as validating the technique on projects from different organizations (cross-organizational validation).

Aiming to address the issues pointed out by [29], performance evaluation metrics more specific to the context of the software testing domain were refined (from the previous study) and proposed.

Furthermore, although the study aimed to validate that the ML model can be interchangeable between languages, a very small number of languages were used, and they have similarities (C, C++, Java). Despite these similarities, Java has considerable differences, for example, in memory management and exception handling, indicating promising results in generalization across different languages but cannot be generalized without further studies. Therefore, new explorations with additional software development projects based on other programming languages with more significant structural differences should be the target of future investigations.

Validating the results for RF is an important contribution since RF has advantages over MLP, the ML approach used in the original study. It is more adaptable to different data types and manages missing values better. Also, it requires less data preprocessing, such as data scaling. It can efficiently process large datasets, requiring less computation, making its training process orders of magnitude faster and cheaper than MLP or more complex ANNs. It offers very rapid predictions after training. It is suitable for solving complex nonlinear relationships between the target and independent variables. Moreover, as demonstrated in this study, where default parameters were used, RF can achieve good results, even without good hyperparameter tuning, which are often comparable to those achieved by well-tuned MLPs. Therefore, RF usage reduces the need for an ML expert. Finally, RF can achieve high accuracies even with smaller data samples, which is crucial to the present application domain since many software development projects are small or medium, and there may not be a high volume of historical data about static source-code metrics and defects. Thus, RF can reduce the entry barrier for software testing informed by ML models.

However, it is noteworthy that although RFs tend to be more resistant to overfitting compared to individual DTs or MLPs because they average multiple decision trees and introduce randomness through bootstrap sampling and feature selection, they are not entirely immune to it. Thus, additional explorations using different datasets with possible distinct noise patterns and ML techniques to validate the current study results will be performed in future studies.

Although the proposed approach uses ML model prediction to inform decision-making on software testing scoping, an essential aspect of ML is the potential value added with its explainability and interpretability. By using RF rather than MLP, the present study also gave an additional step towards a better explainability and interpretability of the cost-sensitive trained models. Higher explainability and interpretability can better inform the software development and quality assurance managers about the main contributing features or source-code characteristics related to software defects, instrumenting them to act on the software development teams to improve the quality of their deliveries in a continuous quality improvement framework. Since enhancing the explainability and interpretability of the cost-sensitive approach can expand its utility for software quality assurance, future studies on this topic are highly recommended.

The research also explored a more comprehensive range [1, 100] of costs (RC) associated with FN compared to the original study [1, 10]. With that, it was possible to observe an asymptotic behavior in the plots of most analyzed metrics. The effect of the marginal gains decreasing with RC increments indicates the cost-sensitive approach reduces its advantages as the test scope is broadened. Thus, as the software testing scope reduction decreases, becoming closer to a complete test, the cost-sensitive approach exhausts its advantages. It is noteworthy that this range may change the observed effects based on the ML algorithm used to induce the ML model; therefore, future investigations must explore distinct RC ranges for different ML algorithms.

The results indicate the possibility of using historical data from previous projects inside the organization combined with the current one at its beginning when almost no historical data is available yet. That enables the early use of ML models to inform software testing scope. However, compared to the benchmark study's findings, the desired positive effects were smoother in the current research. Although the reason is still unknown, when considerable historical data about a system under development or maintenance is already available, it may be better to use the cost-sensitive approach based on a single system's own historical data. The reason for that difference will be the subject of a future study, which will also explore some of the limitations of the current one, such as evaluating other types of ML models, such as Bayesian, meta, tree-based, rule-based, and function-based classifiers.

Finally, the novel dataset ("NASA MDP CROSS-PROJECTS DATASET"), merging all NASA MDP projects and encompassing all their common source-code static features, will be made accessible to the research community. This initiative aims to facilitate further investigations into the effects of cross-language and cross-project dynamics, enabling broader exploration and analysis of the generalization process within the software defect prediction domain.

Author Contributions: Conceptualization, A.M.N.; Methodology, A.M.N.; Project administration, A.M.N. and L.A.V.D.; Software, A.M.N.; Supervision, A.M.N. and L.A.V.D.; Visualization, A.M.N. and G.K.G.S.; Writing—original draft, A.M.N., G.K.G.S. and L.A.V.D.; Writing—review and editing, A.M.N., G.K.G.S. and L.A.V.D.; Funding acquisition, A.M.N. and G.K.G.S. All authors have read and agreed to the published version of the manuscript.

Funding: The APC was funded by 2 A.M. Treinamento e Consultoria Ltda.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The original contributions presented in the study are included in the article, further inquiries can be directed to the corresponding author.

Conflicts of Interest: The authors declare that this study received funding from 2 A.M. Treinamento e Consultoria Ltda. The funder was not involved in the study design, collection, analysis, interpretation of data, the writing of this article or the decision to submit it for publication.

References

1. Cohane, R. Financial Cost of Software Bugs. Available online: https://medium.com/@ryancohane/financial-cost-of-software-bugs-51b4d193f107 (accessed on 31 March 2024).
2. Krasner, H. The Cost of Poor Software Quality in the US: A 2020 Report. *Proc. Consortium Inf. Softw. QualityTM (CISQTM)* **2021**, 1–46. Available online: https://www.it-cisq.org/cisq-files/pdf/CPSQ-2020-report.pdf (accessed on 13 April 2024).
3. Schlappig, B. Regulators Discover New 737 MAX Autopilot Problem. Available online: https://onemileatatime.com/737-max-autopilot-problems/ (accessed on 31 March 2024).
4. Al Root. Boeing Stock's $29 Billion in Lost Value Tells a Story About Earnings. Available online: https://www.barrons.com/articles/boeing-stock-crash-market-value-earnings-51552425879 (accessed on 31 March 2024).
5. Rivero, N. Everything We Know about the Boeing 737 Max 8 Crisis. Available online: https://qz.com/1578227/everything-we-know-about-the-boeing-737-max-8-crashes (accessed on 31 March 2024).
6. Moreira Filho, T.R.; Rios, E. *Projeto & Engenharia de Software: Teste de Software*; ALTA BOOKS: San Francisco, CA, USA, 2003.
7. Alyahya, S. Collaborative Crowdsourced Software Testing. *Electronics* **2022**, *11*, 3340. [CrossRef]
8. Zeng, F.; Liu, S.; Yang, F.; Xu, Y.; Zhou, G.; Xuan, J. Learning to Prioritize Test Cases for Computer Aided Design Software via Quantifying Functional Units. *Appl. Sci.* **2022**, *12*, 10414. [CrossRef]
9. Khatibsyarbini, M.; Isa, M.A.; Jawawi, D.N.A.; Hamed, H.N.A.; Suffian, M.D.M. Test Case Prioritization Using Firefly Algorithm for Software Testing. *IEEE Access* **2019**, *7*, 132360–132373. [CrossRef]
10. Software testing techniques (2nd Edn). B. Beizer, Published by Van Nostrand Reinhold, New York, 1990. ISBN 0-442-20672-0, 550 pages. Price: £36.50, Hard Cover. *Softw. Test. Verif. Reliab.* **1992**, *2*, 215–216. [CrossRef]
11. Evans, I. A Practitioner's Guide to Software Test Design. By Lee Copeland. Published by Artech House, Norwood, MA, U.S.A., 2004. ISBN: 1-58053-791-X, 320 pages. *Softw. Test. Verif. Reliab.* **2004**, *14*, 283–284. [CrossRef]
12. Myers, G.J.; Thomas, T.M.; Sandler, C. *The Art of Software Testing*, 3rd ed.; Wily: Hoboken, NJ, USA, 2011; Volume 1.
13. Li, K.; Wu, M. *Effective Software Test Automation: Developing an Automated Software Testing Tool*; John Wiley & Sons: Hoboken, NJ, USA, 2006.
14. bin Ali, N.; Engström, E.; Taromirad, M.; Mousavi, M.R.; Minhas, N.M.; Helgesson, D.; Kunze, S.; Varshosaz, M. On the search for industry-relevant regression testing research. *Empir. Softw. Eng.* **2019**, *24*, 2020–2055. [CrossRef]
15. Jamil, M.A.; Nour, M.K.; Alotaibi, S.S.; Hussain, M.J.; Hussaini, S.M.; Naseer, A. Software Product Line Maintenance Using Multi-Objective Optimization Techniques. *Appl. Sci.* **2023**, *13*, 9010. [CrossRef]
16. Leicht, N.; Blohm, I.; Leimeister, J.M. Leveraging the Power of the Crowd for Software Testing. *IEEE Softw.* **2017**, *34*, 62–69. [CrossRef]
17. Lachmann, R. 12.4—Machine Learning-Driven Test Case Prioritization Approaches for Black-Box Software Testing. In Proceedings of the ettc2018—European Test and Telemetry Conference, Nürnberg, Germany, 26–28 June 2018.
18. Rätzmann, M.; De Young, C. *Software Testing and Internationalization*; Lemoine International, Incorporated: Salt Lake City, UT, USA, 2003.
19. Broekman, B.; Notenboom, E. *Testing Embedded Software*; Pearson Education: London, UK, 2003.
20. Grbac, T.G.; Runeson, P.; Huljenic, D. A Second Replicated Quantitative Analysis of Fault Distributions in Complex Software Systems. *IEEE Trans. Softw. Eng.* **2012**, *39*, 462–476. [CrossRef]
21. Mauša, G.; Grbac, T.G. Co-evolutionary multi-population genetic programming for classification in software defect prediction: An empirical case study. *Appl. Soft Comput.* **2017**, *55*, 331–351. [CrossRef]
22. Nascimento, A.M.; de Melo, V.V.; Dias, L.A.V.; da Cunha, A.M. Increasing the Prediction Quality of Software Defective Modules with Automatic Feature Engineering. In *Information Technology-New Generations: 15th International Conference on Information Technology*; Springer: Berlin/Heidelberg, Germany, 2018; Volume 738.
23. Elish, K.O.; Elish, M.O. Predicting defect-prone software modules using support vector machines. *J. Syst. Softw.* **2008**, *81*, 649–660. [CrossRef]
24. Li, J.; He, P.; Zhu, J.; Lyu, M.R. Software Defect Prediction via Convolutional Neural Network. In Proceedings of the 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS 2017), Prague, Czech Republic, 25–29 July 2017.
25. Ordonez, M.J.; Haddad, H.M. The State of Metrics in Software Industry. In Proceedings of the International Conference on Information Technology: New Generations (ITNG 2008), Las Vegas, NE, USA, 7–9 April 2008.
26. Shiva, S.G.; Shala, L.A. Software Reuse: Research and Practice. In Proceedings of the International Conference on Information Technology-New Generations (ITNG 2007), Las Vegas, NE, USA, 2–4 April 2007.
27. Zhang, H.; Zhang, X.; Gu, M. Predicting Defective Software Components from Code Complexity Measures. In Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007), Melbourne, Australia, 17–19 December 2007.

28. Alqarni, A.; Aljamaan, H. Leveraging Ensemble Learning with Generative Adversarial Networks for Imbalanced Software Defects Prediction. *Appl. Sci.* **2023**, *13*, 13319. [CrossRef]
29. Babatunde, A.N.; Ogundokun, R.O.; Adeoye, L.B.; Misra, S. Software Defect Prediction Using Dagging Meta-Learner-Based Classifiers. *Mathematics* **2023**, *11*, 2714. [CrossRef]
30. Balogun, A.O.; Basri, S.; Mahamad, S.; Abdulkadir, S.J.; Almomani, M.A.; Adeyemo, V.E.; Al-Tashi, Q.; Mojeed, H.A.; Imam, A.A.; Bajeh, A.O. Impact of Feature Selection Methods on the Predictive Performance of Software Defect Prediction Models: An Extensive Empirical Study. *Symmetry* **2020**, *12*, 1147. [CrossRef]
31. Balogun, A.O.; Basri, S.; Mahamad, S.; Abdulkadir, S.J.; Capretz, L.F.; Imam, A.A.; Almomani, M.A.; Adeyemo, V.E.; Kumar, G. Empirical Analysis of Rank Aggregation-Based Multi-Filter Feature Selection Methods in Software Defect Prediction. *Electronics* **2021**, *10*, 179. [CrossRef]
32. Balogun, A.O.; Basri, S.; Abdulkadir, S.J.; Hashim, A.S. Performance Analysis of Feature Selection Methods in Software Defect Prediction: A Search Method Approach. *Appl. Sci.* **2019**, *9*, 2764. [CrossRef]
33. Khurma, R.; Alsawalqah, H.; Aljarah, I.; Elaziz, M.; Damaševičius, R. An Enhanced Evolutionary Software Defect Prediction Method Using Island Moth Flame Optimization. *Mathematics* **2021**, *9*, 1722. [CrossRef]
34. Pan, C.; Lu, M.; Xu, B.; Gao, H. An Improved CNN Model for Within-Project Software Defect Prediction. *Appl. Sci.* **2019**, *9*, 2138. [CrossRef]
35. Rath, S.K.; Sahu, M.; Das, S.P.; Bisoy, S.K.; Sain, M. A Comparative Analysis of SVM and ELM Classification on Software Reliability Prediction Model. *Electronics* **2022**, *11*, 2707. [CrossRef]
36. Tong, H.; Wang, S.; Li, G. Credibility Based Imbalance Boosting Method for Software Defect Proneness Prediction. *Appl. Sci.* **2020**, *10*, 8059. [CrossRef]
37. Wu, Y.; Yao, J.; Chang, S.; Liu, B. LIMCR: Less-Informative Majorities Cleaning Rule Based on Naïve Bayes for Imbalance Learning in Software Defect Prediction. *Appl. Sci.* **2020**, *10*, 8324. [CrossRef]
38. Shepperd, M.; Song, Q.; Sun, Z.; Mair, C. Data Quality: Some Comments on the NASA Software Defect Datasets. *IEEE Trans. Softw. Eng.* **2013**, *39*, 1208–1215. [CrossRef]
39. Gray, D.; Bowes, D.; Davey, N.; Sun, Y.; Christianson, B. The Misuse of the NASA Metrics Data Program Data Sets for Automated Software Defect Prediction. In Proceedings of the IET Seminar Digest, Durham, UK, 11–12 April 2011; Volume 2011.
40. Moreira Nascimento, A.; Vismari, L.F.; Cugnasca, P.S.; Camargo Junior, J.B.; Rady De Almeira Junior, J. A Cost-Sensitive Approach to Enhance the Use of ML Classifiers in Software Testing Efforts. In Proceedings of the 18th IEEE International Conference on Machine Learning and Applications (ICMLA 2019), Boca Raton, FL, USA, 16–19 December 2019.
41. Freiesleben, T.; Grote, T. Beyond generalization: A theory of robustness in machine learning. *Synthese* **2023**, *202*, 1–28. [CrossRef]
42. Maleki, F.; Ovens, K.; Gupta, R.; Reinhold, C.; Spatz, A.; Forghani, R. Generalizability of Machine Learning Models: Quantitative Evaluation of Three Methodological Pitfalls. *Radiol. Artif. Intell.* **2023**, *5*, e220028. [CrossRef]
43. Verkerken, M.; D'hooge, L.; Wauters, T.; Volckaert, B.; De Turck, F. Towards Model Generalization for Intrusion Detection: Unsupervised Machine Learning Techniques. *J. Netw. Syst. Manag.* **2022**, *30*, 12. [CrossRef]
44. Sankaranarayanan, S.; Balaji, Y.; Jain, A.; Lim, S.N.; Chellappa, R. Learning from Synthetic Data: Addressing Domain Shift for Semantic Segmentation. In Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition, Salt Lake City, UT, USA, 18–22 June 2018.
45. Langford, M.A.; Cheng, B.H.C. "know What You Know": Predicting Behavior for Learning-Enabled Systems When Facing Uncertainty. In Proceedings of the 2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2021), Madrid, Spain, 18–24 May 2021.
46. Feng, S.; Keung, J.; Yu, X.; Xiao, Y.; Zhang, M. Investigation on the stability of SMOTE-based oversampling techniques in software defect prediction. *Inf. Softw. Technol.* **2021**, *139*, 106662. [CrossRef]
47. Pachouly, J.; Ahirrao, S.; Kotecha, K.; Selvachandran, G.; Abraham, A. A systematic literature review on software defect prediction using artificial intelligence: Datasets, Data Validation Methods, Approaches, and Tools. *Eng. Appl. Artif. Intell.* **2022**, *111*, 104773. [CrossRef]
48. Hall, T.; Beecham, S.; Bowes, D.; Gray, D.; Counsell, S. A Systematic Literature Review on Fault Prediction Performance in Software Engineering. *IEEE Trans. Softw. Eng.* **2012**, *38*, 1276–1304. [CrossRef]
49. Abdolrasol, M.G.M.; Hussain, S.M.S.; Ustun, T.S.; Sarker, M.R.; Hannan, M.A.; Mohamed, R.; Ali, J.A.; Mekhilef, S.; Milad, A. Artificial Neural Networks Based Optimization Techniques: A Review. *Electronics* **2021**, *10*, 2689. [CrossRef]
50. Panerati, J.; Schnellmann, M.A.; Patience, C.; Beltrame, G.; Patience, G.S. Experimental methods in chemical engineering: Artificial neural networks–ANNs. *Can. J. Chem. Eng.* **2019**, *97*, 2372–2382. [CrossRef]
51. Chen, S.; Ren, Y.; Friedrich, D.; Yu, Z.; Yu, J. Sensitivity analysis to reduce duplicated features in ANN training for district heat demand Prediction. *Energy AI* **2020**, *2*, 100028. [CrossRef]
52. Zhou, T.; Wang, F.; Yang, Z. Comparative Analysis of ANN and SVM Models Combined with Wavelet Preprocess for Groundwater Depth Prediction. *Water* **2017**, *9*, 781. [CrossRef]
53. Shah, D.; Wang, J.; He, Q.P. Feature engineering in big data analytics for IoT-enabled smart manufacturing—Comparison between deep learning and statistical learning. *Comput. Chem. Eng.* **2020**, *141*, 106970. [CrossRef]
54. Chu, J.; Liu, X.; Zhang, Z.; Zhang, Y.; He, M. A novel method overcomeing overfitting of artificial neural network for accurate prediction: Application on thermophysical property of natural gas. *Case Stud. Therm. Eng.* **2021**, *28*. [CrossRef]

55. Lin, C.-J.; Wu, N.-J. An ANN Model for Predicting the Compressive Strength of Concrete. *Appl. Sci.* **2021**, *11*, 3798. [CrossRef]
56. Adadi, A.; Berrada, M. Peeking Inside the Black-Box: A Survey on Explainable Artificial Intelligence (XAI). *IEEE Access* **2018**, *6*, 52138–52160. [CrossRef]
57. Razavi, S. Deep learning, explained: Fundamentals, explainability, and bridgeability to process-based modelling. *Environ. Model. Softw.* **2021**, *144*, 105159. [CrossRef]
58. Chapman, M.; Callis, P.; Menzies, T. JM1/Software Defect Prediction 2004. Available online: http://promise.site.uottawa.ca/SERepository/datasets/jm1.arff (accessed on 13 April 2024).
59. Ma, Y.; Luo, G.; Zeng, X.; Chen, A. Transfer learning for cross-company software defect prediction. *Inf. Softw. Technol.* **2012**, *54*, 248–256. [CrossRef]
60. Alsghaier, H.; Akour, M. Software fault prediction using Whale algorithm with genetics algorithm. *Softw. Pract. Exp.* **2021**, *51*, 1121–1146. [CrossRef]
61. Gaffney, J.E., Jr. Metrics in Software Quality Assurance. In Proceedings of the ACM'81 conference, Los Angeles, CA, USA, 9–11 November 1981; pp. 126–130.
62. Halstead, M.H. Toward a theoretical basis for estimating programming effort. In Proceedings of the 1975 Annual Conference, Minneapolis, MV, USA, 20–22 October 1975; pp. 222–224. [CrossRef]
63. McCabe, T.J.; Butler, C.W. Design complexity measurement and testing. *Commun. ACM* **1989**, *32*, 1415–1425. [CrossRef]
64. McCabe, T.J. A Complexity Measure. *IEEE Trans. Softw. Eng.* **1976**, *SE 2*, 308–320. [CrossRef]
65. Wilson, D.L. Asymptotic Properties of Nearest Neighbor Rules Using Edited Data. *IEEE Trans. Syst. Man Cybern.* **1972**, *2*, 408–421. [CrossRef]
66. Lewis, D.D.; Catlett, J. Heterogeneous Uncertainty Sampling for Supervised Learning. In Proceedings of the 11th International Conference on Machine Learning (ICML 1994), New Brunswick, NJ, USA, 10–13 July 1994.
67. Kubat, M.; Matwin, S. Addressing the Curse of Imbalanced Training Sets: One-Sided Selection. In Proceedings of the International Conference on Machine Learning, Nashville, TN, USA, 8–12 July 1997.
68. Chambers, R.L. Robust Case-Weighting for Multipurpose Establishment Surveys. *J. Off. Stat.-Stockh.* **1996**, *12*, 3–32.
69. Chawla, N.V.; Bowyer, K.W.; Hall, L.O.; Kegelmeyer, W.P. SMOTE: Synthetic Minority Over-sampling Technique. *J. Artif. Intell. Res.* **2002**, *16*, 321–357. [CrossRef]
70. Nascimento, A.M.; de S. Meirelles, F. An Artificial Intelligence Adoption Intention Model (AI2M) Inspired by UTAUT. In Proceedings of the Information Systems in Latin America (ISLA 2022), Virtually, 8–10 August 2022.
71. Hall, M.; Frank, E.; Holmes, G.; Pfahringer, B.; Reutemann, P.; Witten, I.H. The WEKA Data Mining Software: An Update. *ACM SIGKDD Explor. Newsl.* **2009**, *11*, 10–18. [CrossRef]
72. Breiman, L. Random forests. *Mach. Learn.* **2001**, *45*, 5–32. [CrossRef]
73. Rodriguez-Galiano, V.; Sanchez-Castillo, M.; Chica-Olmo, M.; Chica-Rivas, M. Machine learning predictive models for mineral prospectivity: An evaluation of neural networks, random forest, regression trees and support vector machines. *Ore Geol. Rev.* **2015**, *71*, 804–818. [CrossRef]
74. Liaw, A.; Wiener, M. Classification and Regression by RandomForest. *R News* **2002**, *2*, 18–22.
75. Robinson, R.L.M.; Palczewska, A.; Palczewski, J.; Kidley, N. Comparison of the Predictive Performance and Interpretability of Random Forest and Linear Models on Benchmark Data Sets. *J. Chem. Inf. Model.* **2017**, *57*, 1773–1792. [CrossRef]
76. Kumaravel, A.; Vijayan, T. Comparing cost sensitive classifiers by the false-positive to false-negative ratio in diagnostic studies. *Expert Syst. Appl.* **2023**, *227*, 120303. [CrossRef]
77. Meekins, R.; Adams, S.; Beling, P.A.; Farinholt, K.; Hipwell, N.; Chaudhry, A.; Polter, S.; Dong, Q. Cost-Sensitive Classifier Selection When There is Additional Cost Information. In Proceedings of the Machine Learning Research, 2018; Volume 88. Available online: https://proceedings.mlr.press/v88/meekins18a.html (accessed on 13 April 2024).
78. Stone, M. Cross-Validatory Choice and Assessment of Statistical Predictions. *J. R. Stat. Soc. Ser. B (Methodol.)* **1974**, *36*, 111–133. [CrossRef]
79. Kohavi, R. A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection. In Proceedings of the IJCAI International Joint Conference on Artificial Intelligence, Montreal, QC, Canada, 20–25 August 1995; Volume 2.
80. Menzies, T.; Di Stefano, J.S. How Good Is Your Blind Spot Sampling Policy. In Proceedings of the Eighth IEEE International Symposium on High Assurance Systems Engineering, Tampa, FL, USA, 25–26 March 2004; pp. 129–138.
81. Seliya, N.; Khoshgoftaar, T.M.; Van Hulse, J. A Study on the Relationships of Classifier Performance Metrics. In Proceedings of the International Conference on Tools with Artificial Intelligence (ICTAI), Newark, NJ, USA, 2–4 November 2009.