Article

Not peer-reviewed version

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

Alexandre Moreira Nascimento * , Gabriel Kenji Godoy Shimanuki , Luiz Alberto Vieira Dias

Posted Date: 16 May 2024

doi: 10.20944/preprints202405.1028.v1

Keywords: Machine Learning; Imbalance; Software Defect Prediction; NASA MDP; Random Forest; Software Quality; Generalization; Cost-Sensitive; Cross-language; Cross-Project

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

[1] Stanford University; alexandremoreiranascimento@alum.mit.edu
[2] Universidade de São Paulo (USP); gabrielshimanuki@usp.br
[3] Instituto Tecnológico de Aeronáutica (ITA); vdias@ita.br
* Correspondence: alexandremoreiranascimento@alum.mit.edu

Featured Application: The technique uses Machine Learning (ML) models to support decision-making on software testing scope and resources 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, the generalizability of these models across different projects (cross-project) and programming languages (cross-language) remained untested. This study validates the model's applicability across diverse development environments by integrating various datasets from distinct projects into a unified, using a more interpretable ML approach. The results demonstrate that ML can support software testing decisions, enabling teams to identify up to seven times more defective modules with the same testing effort as a benchmark.

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

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 a per capita yearly cost of software issues reaching $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. Illustrated in Figure 1 is a commonly utilized pipeline in software defect prediction projects. ML classifier models have the capability to be trained using a historical dataset containing each module's static source-code metrics and an indication of whether it was defective or not [20–24] to highlight the system modules most prone to defects [25–27]. 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 in a more focused approach.

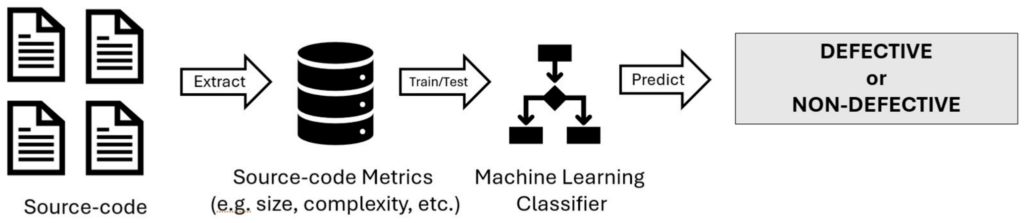


Figure 1. Typical ML training process to create a model to predict defective software modules.

An extensive body of research software defects prediction based on ML models exists. Literature approaches defective prediction models from various angles [28–37]. One of the most well-known datasets used in many of those studies are 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. However, in contrast, [34] exclusively relies on datasets sourced from the PROMISE repository for its research endeavors. As will be shown, the most common techniques to tackle the problem of software prediction datasets are Naïve Bayes (NB) and Decision Tree (DT). [29] concluded that the utilization of dagging-based classifiers enhanced software defect prediction models relative to baseline classifiers like NB, DT, and k-Nearest Neighbor (kNN). Additionally, the study noted that the efficacy of machine learning algorithms can vary depending on the performance metrics employed and the specific conditions of the experiment. [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, 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. [34] proposes a new method based on Convolutional Neural Networks (CNNs) to identify patterns associated to bugs. Despite the good results presented, it is shown that "the technique exhibited a negative response to hyperparameter instability" [34]. Put differently, applying this type of model in real-life scenarios could result in "a fluctuation when focused on an individual version or project" [34]. Nonetheless, as mentioned in text, it's worth noting that instability may predict various types of defects simply by adjusting hyperparameters in alternative ways.

Attempting to address the issue from another perspective, some studies focus on the quality of the data utilized by the models. [37] proposes a resampling method utilizing NB, but it fails to outperform across all datasets, highlighting that there are no universally effective imbalance learning methods; thus, selecting appropriate methods is crucial. [36] proposes a method and compares it with existing ones, addressing issues with imbalanced learning such as interference with real data caused by the use of SMOTE, emphasizing the importance of focusing on the data quality of synthetic data. Lastly, in addressing the imbalance problem, [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 like SMOTE, ROS, and RUS. However, it is important to note that the combination of undersampling techniques with GANs may result in a degradation in prediction performance due to the elimination of 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.

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

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 should be considered by compensating for the imbalance with one of the existing techniques. However, among 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 a non-existing high quality of the software, misleading managers to lower the software testing efforts and deflecting the efforts from those many misclassified defective modules. Accordingly, those issues remain in the software, resulting in future operational failures that could lead to severe consequences.

Previous research proposed and evaluated a novel approach to support managers in making better decisions to optimize the outcomes of a software testing effort. Studies proposed novel techniques to enhance the learning of the ML model. For example, research [27] 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 that the software testing scope was most prone to defects. 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.

A method leveraged the dataset imbalance and cost-sensitive ML training to improve the ML model results, considering resource availability and smoothing unwanted FN effects. Using cost-sensitive ML training, a study [40] demonstrated an approach to handle dataset imbalance in predicting defective software modules. Based on adjusting the costs 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 ML model's generalizability in cross-project and cross-language scenarios.

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, generalizability also 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. However, a cross-project and cross-language generalizable ML model could be a game changer. That ML model, trained with data from other systems 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 usefulness of those ML model-based techniques.

Another limitation of the study using cost-sensitive ML training to support software testing decision-making [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, long training times, and suitable hardware due to their high computational cost, which may not be available [44–46]. They also require more challenging data preprocessing, feature engineering, and hyperparameters tuning, which may require a specialization beyond the conventional software testing staff [47,48]. 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 [49,50]. Finally, its black-box nature makes its explainability and interpretability poor [51,52]. 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 better decision-making on what to work on 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 based on different programming languages. Furthermore, the present work expanded the investigation, using a dataset almost 4.5 times larger than the baseline study [40]. To our knowledge, no other study has used the proposed approach in the defect prediction domain and validated its generalization ability in the way executed here.

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 for evaluating the experiment is explained. Finally, the evaluation metrics used to support the analysis are detailed.

2.1. Dataset

NASA opened 14 datasets regarding distinct software development projects to support research on software module defect prediction [53]. The datasets cover 14 software development projects based on various programming languages. Each dataset instance 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 [54–57]. 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 [58] 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	**C**	**J**	**K**	**K**	**MC**	**M**	**M**	**P**	**PC**	**PC**	**PC**	**PC**
Number of Instances	34	95	20	20	8737	12	263	73	14	10	13	169
Number of Features	37	21	21	39	38	39	37	37	36	37	37	38
Programming Language	C	C	C+	Ja	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_COMM	**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_COMPLEXI	**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_SEVERIT	x			x	x	x	x	x	x	x	x	x
*MODIFIED_CONDITION_*	x			x	x	x	x	x	x	x	x	x

MULTIPLE_CONDITION_C	x			x	x	x	x	x	x	x	x	x
NODE_COUNT	x			x	x	x	x	x	x	x	x	x
NORMALIZED_CYLOMAT	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_OPERAND	X	X	X	X	X	X	X	X	X	X	X	X
NUM_UNIQUE_OPERATO	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_COMPLE				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 generalizability. Its assembly was done by carefully merging the 12 NASA datasets. From the original NASA database, KC2 and KC4 were excluded due to significant discrepancies in their features, which could potentially lead to problems in experiments. However, the slight difference in each dataset's number of features imposes some challenges in this merging process, which 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 cross-project and cross-language generalizability. The features "common denominator" among all datasets were identified to overcome this challenge, as shown in Table 1. As a result, a total of 20 features were identified as present in all datasets marked with a light gray background in Table 1. A newer and compatible version of all the datasets was created by removing all the features that did not belong to this set of features obtained with the intersection of the dataset's common features. Then, all those new compatible datasets were merged into a 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 9,593 instances with 21 features of a single project's software modules in C language, the assembled dataset contains 43023 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 [59], under-sampling [60,61], case weighting [62], or synthetic minority oversampling technique (SMOTE) [63], was used to balance the dataset classes to follow the same protocol used by the original research.

2.2. Machine Learning 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], aiming to avoid the ANNs' weakness, 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 [64].

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 [65]. It also requires shorter training times and less advanced hardware for training [66]. 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 [65].

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 [64]. 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 [67]. Feature importance scores can be generated, offering insights into model decisions, and supporting managers' decision-making on policies and actions to improve the software quality in future development iterations.

2.3. Experimental Protocol

The same original study [40] protocol was used 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 effect of the unbalanced dataset in generating the worrisome FNs. 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), aiming 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 set of costs 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. Since, as in [68,69], 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.

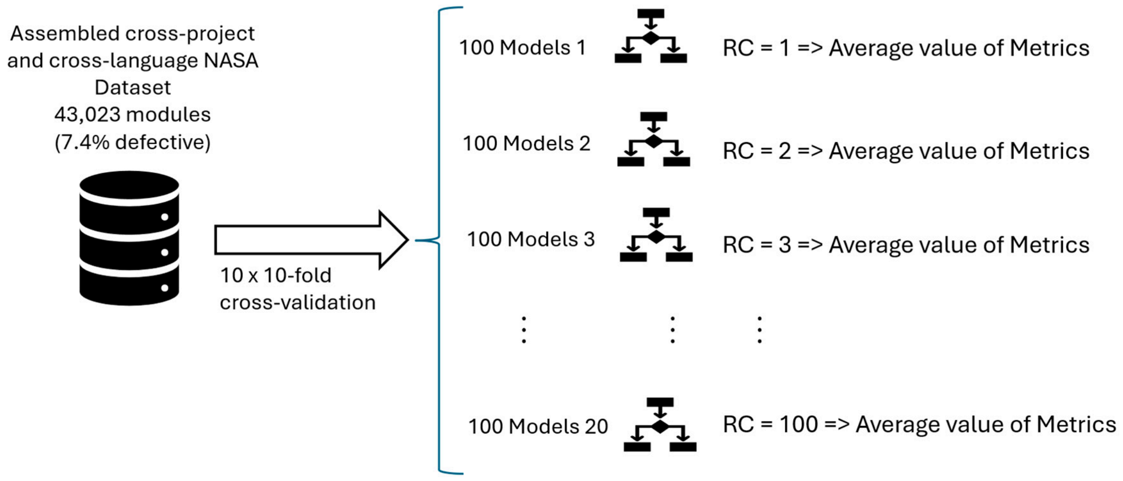

Assembled cross-project and cross-language NASA Dataset
43,023 modules
(7.4% defective)
10 x 10-fold cross-validation
100 Models 1
RC = 1 => Average value of Metrics
100 Models 2
RC = 2 => Average value of Metrics
100 Models 3
RC = 3 => Average value of Metrics
100 Models 20
RC = 100 => Average value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

Figure 2. Cost-sensitive ML training 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 10x10-fold cross-validation strategy [70]. The 10x10-fold cross-validation supports a more reliable validation of the proposed technique. Among the arguments by [71], 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 it results in less variance, which helps to compare the performance of distinct ML models induced as the FN assigned cost is increased. Smoothing out the extreme effects of the luckiest and unluckiest data selection for training and testing 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.

Therefore, 100 validations were performed for each distinct cost assigned to FN. A unitary cost (1) was assigned to TP, TN, and FP for all the ML models in this study. Since 20 distinct FN cost values were used, 2000 RFs were generated. The average value of the evaluation metrics (subsection D) from the 100 RF was computed for each distinct cost value experimented.

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].

$$\mathrm{MT} = \mathrm{TP} + \mathrm{FP}, \tag{1}$$

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)**.

$$\mathrm{MNT} = \mathrm{TN} + \mathrm{FN}, \tag{2}$$

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].

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

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 by Equation 4.

$$\mathrm{RTS} = 1 - \mathrm{SR}, \tag{4}$$

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.

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

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

$$\mathrm{RC} = C_{FN}, \tag{6}$$

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)** [72] 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].

$$Eff = P = \frac{TP}{(TP + FP)}, \quad (7)$$

Analogously, according to [40], **Recall (R)** [72] 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].

$$Ef = R = \frac{TP}{(TP + FN)}, \quad (8)$$

Furthermore, the ML model **Accuracy (Acc)** [73,74] 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 [73,74].

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

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.

$$REffU_{RC} = \frac{Eff_{RC}}{Eff_{RC=1}}, \quad (10)$$

$$REfU_{RC} = \frac{Ef_{RC}}{Ef_{RC=1}}, \quad (11)$$

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 **relative efficiency to the random selection (REffR)** 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 **relative efficacy to the random selection (REfR)** was computed for each RC to support those comparisons using Equation 13.

$$REffR_{RC} = p_{TP} = 7.4\%, \quad (12)$$

$$REfR_{RC} = REffR_{RC} \times RTS_{RC} = 7.4\% \times RTS_{RC}, \quad (13)$$

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 14, while the **Relative Percent Correct relative to the Random selection of modules (RPCR)** was computed using Equation 15.

$$RPCU_{RC} = \frac{ACC_{RC}}{ACC_{RC=1}}, \tag{14}$$

$$RPCR_{RC} = \frac{ACC_{RC}}{7.8\%}, \tag{15}$$

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 16 and 17, respectively.

$$MNM_{RC} = \frac{FN_{RC}}{\frac{3196}{k}}, \tag{16}$$

$$MNDM_{RC} = \frac{FP_{RC}}{\frac{39827}{k}}, \tag{17}$$

Finally, the metric **Unnecessary Tests** (UT) [40] was computed to evaluate the ratio of module tests that were wasted because they were unnecessary. Equation 18 shows how UT was calculated for each value of RC evaluated.

$$UT = \frac{FP_{RC}}{MT_{RC}}, \tag{18}$$

3. Results

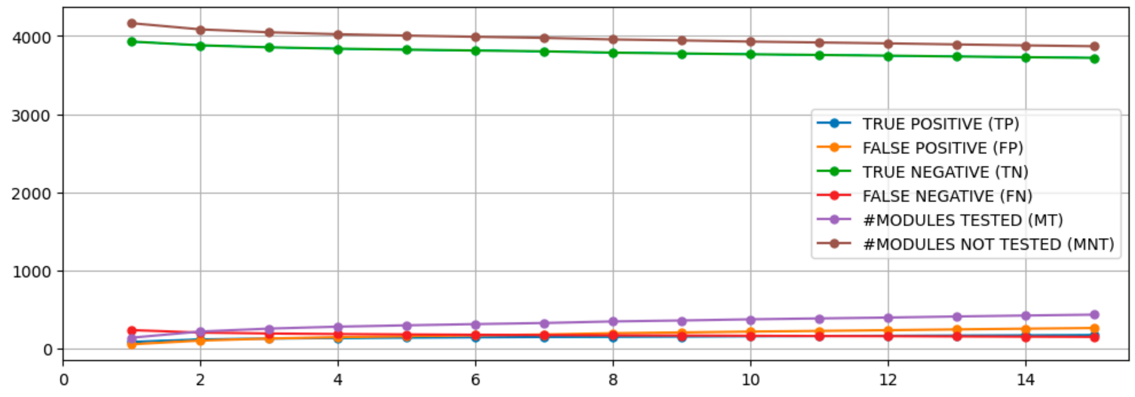
The protocol described in the previous section was executed entirely, providing a dataset of results with the metrics used to support the analysis presented here. Table 2 shows the models' accuracies for each relative cost. A paired t-test (with correction) was used to compare the accuracies' averages 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 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 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).

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

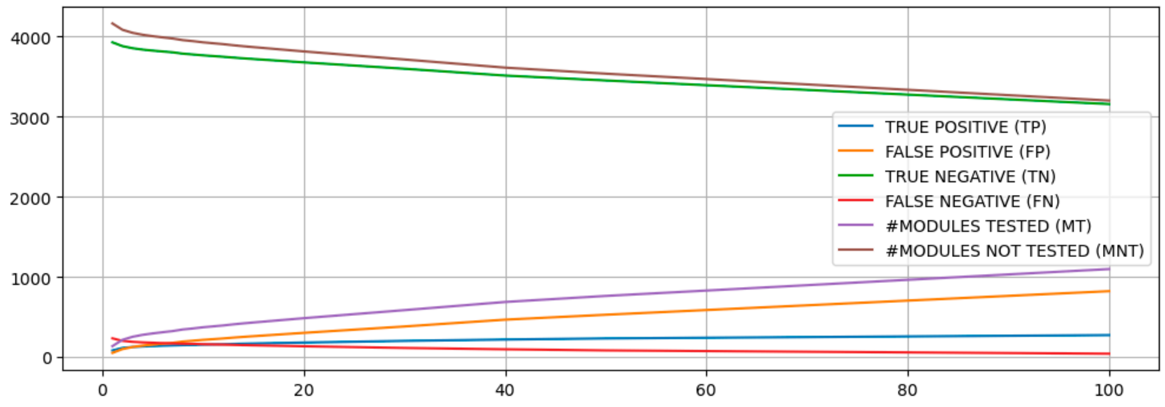
RC	ACC	TP	FP	TN	FN	MT	MNT	SR
1	93.3%	84.04	54.11	3928.59	235.56	138.15	4164.15	96.8%
2	92.9% *	115.65 *	101.62 *	3881.08 *	203.95 *	217.27	4085.03	94.9%
3	92.6% *	127.42 *	127.5 *	3855.2 *	192.18 *	254.92	4047.38	94.1%
4	92.3% *	134.5 *	144.87 *	3837.83 *	185.1 *	279.37	4022.93	93.5%
5	92.2% *	139.28 *	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%

Table 2 also shows the information from the confusion matrix (TP, TN, FP, FN), indicating how its distribution and RC change. 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 over 7 times in the reference study using a different classifier on a single project and language dataset. Naturally, the increase in TP and decrease in FN could only happen with an increase in FP (~4x) 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 (2.7x) 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 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.



(**a**) Initial RC range



(**b**) Full RC range

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

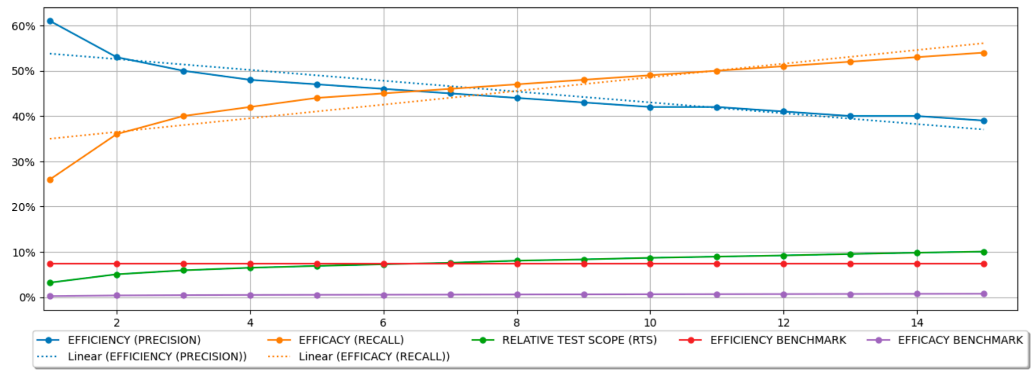
Table 3 shows the classifiers' test efficiency and efficacy metrics and a theoretical benchmark obtained with the expected results from a random selection of modules to be tested with the same scope size for each RC. The TP, TN, FP, and FN changes have essential implications for the efficiency, efficacy, 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 7.6x, 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 1.9x. Finally, a smoother effect was also observed for RTS (an increase of 2.7x) compared to the reference study (an increase of 17.8x).

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

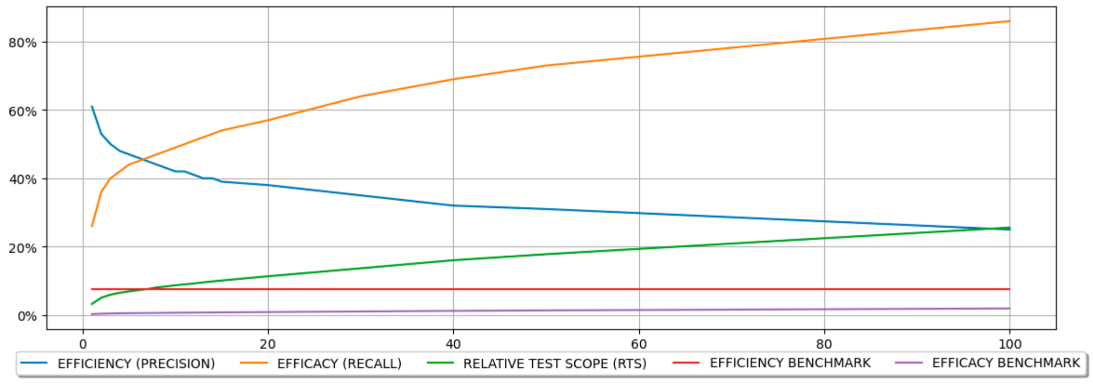
RC	ML Model			Benchmarks							
				Random Selection					Unitary Cost ML Model		
	EFF	EF	RTS	EFF	EF	REFF	RE	RPC	REFF	RE	RPC
1	61.0%	26.0%	3.2%	7.4%	0.2%	821.2%	10899.8%	1255.6%	100.0%	100.0%	100.0%
2	53.0% *	36.0% *	5.1%	7.4%	0.4%	713.5%	9596.2%	1250.6%	86.9%	138.5%	99.6%
3	50.0% *	40.0% *	5.9%	7.4%	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%

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 2.5x 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.



(a) Initial RC range



(**b**) Full RC range

Figure 4. {TP, TN, FP, FN, MT and MNT} x RC.

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 19 and 20.

$$RECALL = 0.00151 \cdot RC + 0.3348;\ R^2 = 0.8394, \tag{19}$$

$$PRECISION = -0.012 \cdot RC + 0.5497;\ R^2 = 0.8322, \tag{20}$$

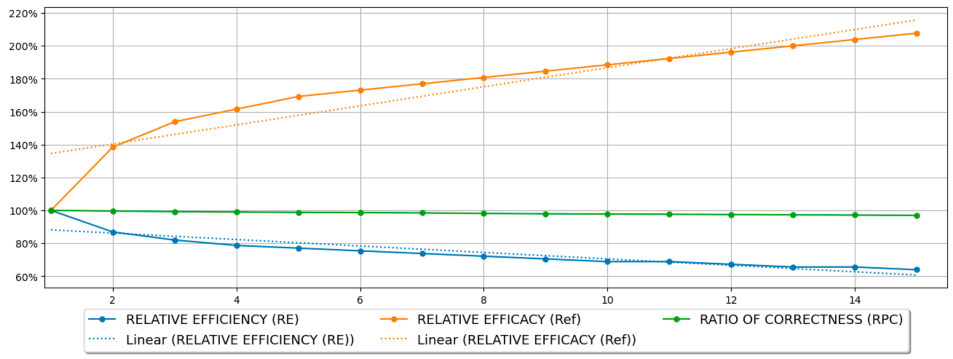
It's important to highlight that all linear regressions conducted in this study were tailored to RC ranges from 1 to 15 for two primary reasons. Firstly, the baseline study [40] had a narrower range of 1 to 10, necessitating an adjustment of parameters to facilitate a more accurate comparison. Secondly, 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.

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.

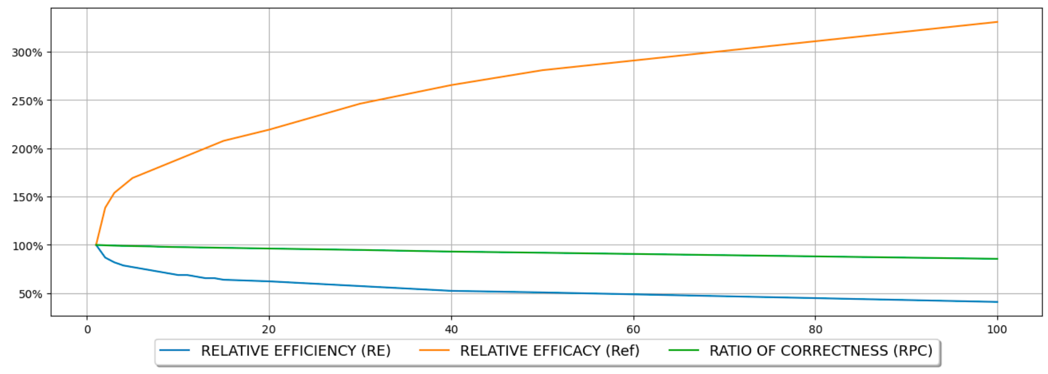
Those results indicate that conveniently adjusting RC makes finding an optimal equilibrium between efficiency and efficacy and the extent of testing coverage possible. This is the core idea of the approach, which has been demonstrated only for a single project dataset until now. Thus, it indicates that 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.

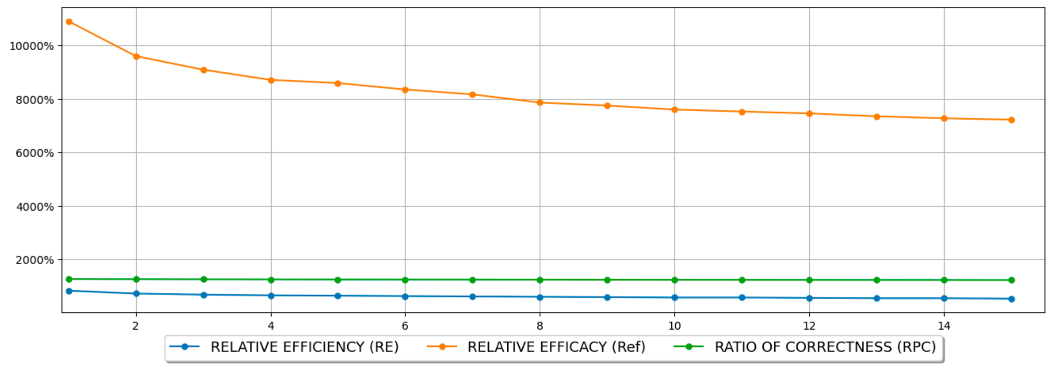
Additionally, Table 3 presents comparisons of efficiency and efficacy with other benchmarks. One of the benchmarks used was the efficiency and efficacy of the unitary cost model ($RC = 1$). Thus, for each RC value, the table shows how the ML model's performance (efficiency and efficacy) compares to the baseline model's performance ($RC = 1$). Table 3 also shows the **relative percent correct (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.



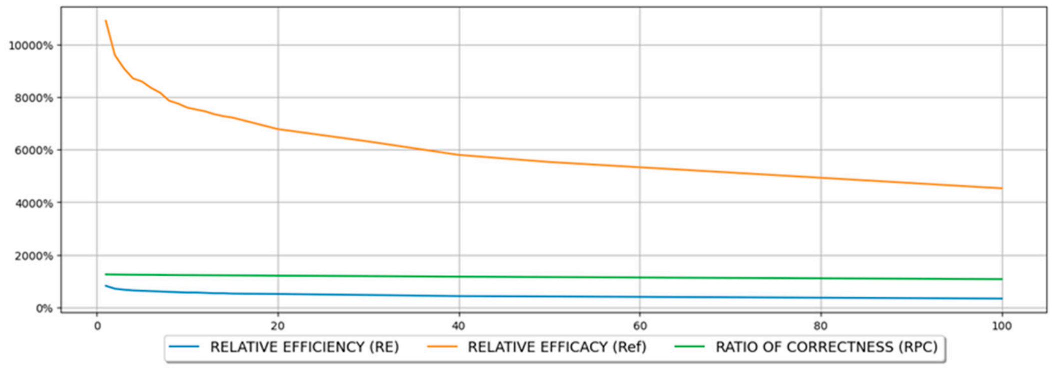
(a) Benchmark: Unitary Cost ML Model – Partial RC range.



(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. RELATIVE EFFICIENCY, EFFICACY, and RPC.

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 21 and 22.

$$Ref \ = \ 0.058 \cdot RC \ + \ 1.2875; \ R^2 = 0.8394, \tag{21}$$

$$RE \ = \ -0.0196 \cdot RC \ + \ 0.9012; \ R^2 = 0.8322, \tag{22}$$

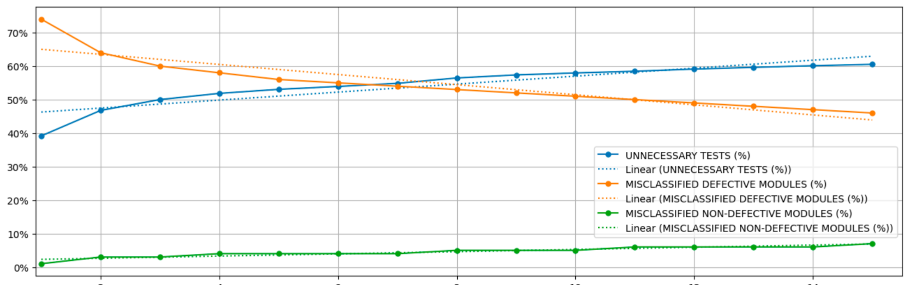
The plots and descriptions of those models are illustrated in Figure 5. Since all the regressions reached $R^2 > 80\%$, they can be considered suitable for explaining relative efficiency and relative efficacy variances by RC changes. Comparing the effect sizes of RC on efficiency (-0.0196) and efficacy (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 3x higher than the efficiency decrease. Thus, each RC unit increment returns an improvement in relative test efficacy 3x 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.

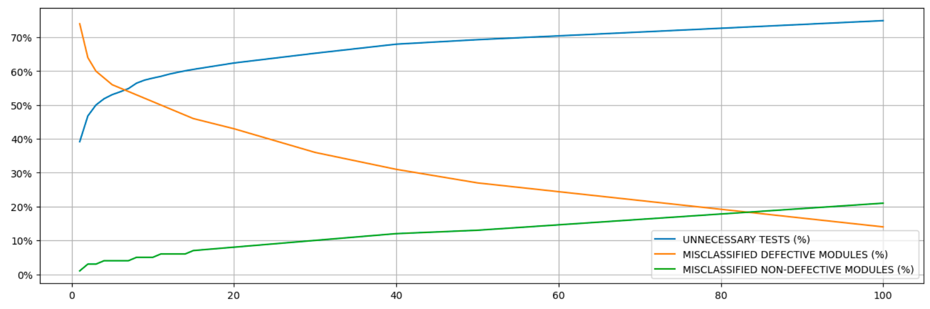
The last analysis was to understand how RC increases affect MDM, MNDM, and UT. Table 4 shows the values of each of those metrics for each RC value. The experiments were conducted using a 10-fold cross-validation approach. Thus, each fold contained on average 320 defective modules (used for MDM calculation, with $k = 10$) 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.

Table 4. MDM, MNDM, and UT x 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%



(**a**) Initial RC range



(**b**) Full RC range

Figure 6. {MDM, MNDM, and UT} x 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 23, 24 and 25.

$$UT = 0.0119 \cdot RC + 0.4509;\ R^2 = 0.8284, \tag{23}$$

$$MDC = -0.0151 \cdot RC + 0.6652;\ R^2 = 0.8394, \tag{24}$$

$$MNDC = 0.0033 \cdot RC + 0.0197;\ R^2 = 0.8997, \tag{25}$$

Those models reached R^2 values 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's 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 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.

That is, the approach is very appealing to inform 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 suits 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 using a hypothetical scenario.

The test manager has a budget of n_t software testers and n_d days available for software testing. Equation 26 gives the available software testing budget in hours (B), considering a workday of 8 hours/day.

$$B = 8 \times n_t \times n_d, \tag{26}$$

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 hours. The project's average effort to test a software module (E) was estimated to be 5 hours. Thus, the available budget can afford to test around 218 modules on average. Since in Table 2, the closest MT value to 218 is 217, which corresponds to $RC = 2$, the ML model to be used to support the decision-making on software testing must be one trained with $RC = 2$.

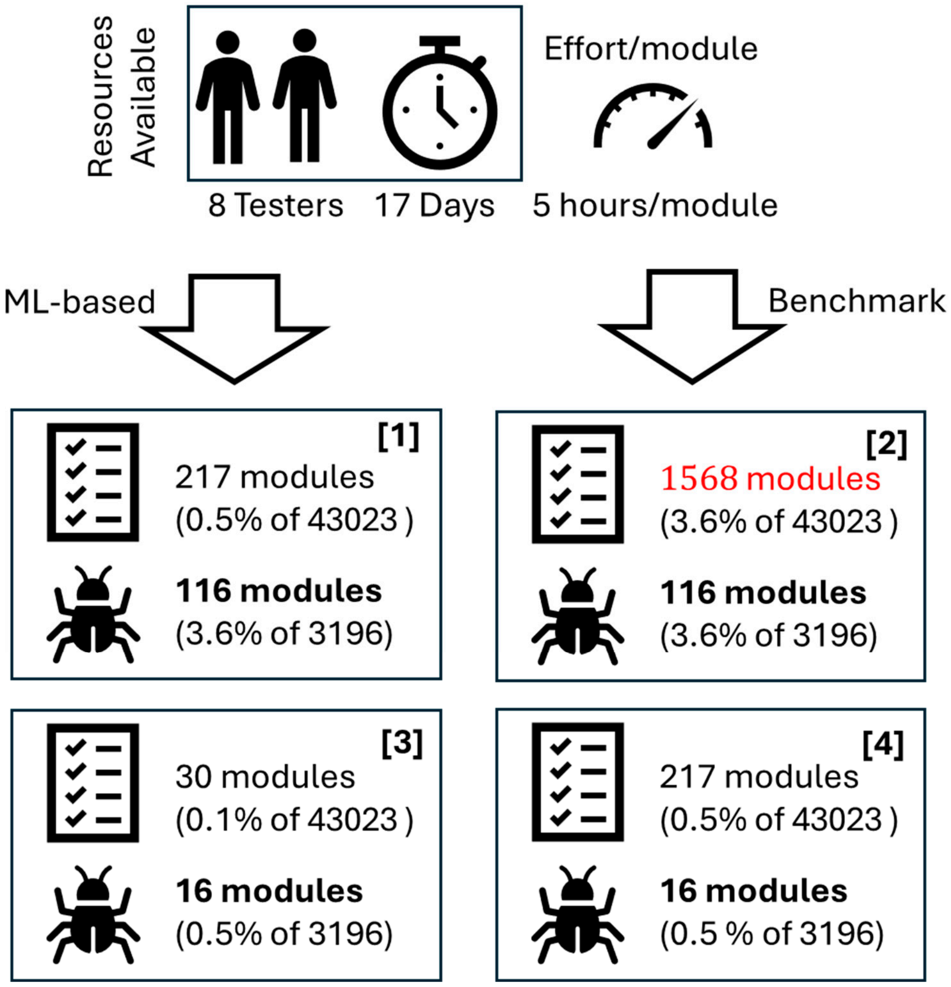


Figure 7. Comparison of the value added by the proposed technique in a hypothetical scenario with at least for decisions regarding the software testing scoping and allocation of available software testing resources.

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 were performed in a same-size scope encompassing modules randomly selected, only 16 defective modules would be discovered (Figure 7.[3]), given by Equation 27.

$$DR = p_{TP} \times \mathrm{MT} = 7.4\% \times 217 = 16, \tag{27}$$

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] x Figure 7.[3]). Moreover, a manager would need a testing effort (TE_{Random}) of 7840 hours (given by Equation 28) to test 1568 software modules randomly picked to accomplish an equivalent performance in the number of defective modules discovered (Figure 7.[1] x 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.

$$TE_{Random} = \frac{TP}{p_{TP}} \times E = \frac{116}{7.4\%} \times 5 = 7840, \tag{28}$$

Considering this hypothetical scenario, following the ML model recommendations, the software testing productivity (Prod) would be 9.4 hours to find each defective module on average, given by Equation 29.

$$Prod = \frac{MT}{TP} \times E = \frac{217}{116} \times 5 = 9.4, \tag{29}$$

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] x Figure 7.[4]), according to Equation 30, which would require a total effort of only 30 x 9.4 = 282 hours.

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

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. 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 using the RF, a distinct ML classifier, a much larger dataset, with a higher imbalance ratio, encompassing multiple software development projects based on different software programming languages. Generalizability validation is one of the relevant contributions of the present research since it demonstrates that the approach can be useful and reliable across various distinct situations involving distinct projects, languages, and teams, indicating it can be effective in practical applications.

Validating the results for RF is an important contribution since RF has advantages over MLP. 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. It does not require an ML expert because even without good hyperparameter tuning, it can achieve good results, which are often comparable to those achieved by well-tuned MLPs. Finally, RF can manage to 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 statical source-code metrics and defects. Thus, RF can reduce the entering barrier for software testing informed by ML models.

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 enhanced the explainability and interpretability of the cost-sensitive trained models, expanding their initial utility. They can 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.

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.

The results show that it is possible to use historical data from previous projects 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, encompassing all common NASA MDP datasets 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: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Ryan Cohane 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.
3. Ben Schlappig 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. Nicolás Rivero 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. Trayahú FILHO, R.M.; RIOS, E. Projeto & Engenharia de Software: Teste de Software 2003.
7. Alyahya, S. Collaborative Crowdsourced Software Testing. *Electronics (Switzerland)* **2022**, *11*, doi:10.3390/electronics11203340.
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. *Applied Sciences (Switzerland)* **2022**, *12*, doi:10.3390/app122010414.
9. Khatibsyarbini, M.; Isa, M.A.; Jawawi, D.N.A.; Hamed, H.N.A.; Mohamed Suffian, M.D. Test Case Prioritization Using Firefly Algorithm for Software Testing. *IEEE Access* **2019**, *7*, doi:10.1109/ACCESS.2019.2940620.

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. *Software Testing, Verification and Reliability* **1992**, *2*, doi:10.1002/stvr.4370020406.
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. *Software Testing, Verification and Reliability* **2004**, *14*, doi:10.1002/stvr.305.
12. Myers, G.J.; Thomas, T.M.; Sandler, C. *The Art of Software Testing 3rd Edition*; 2011; Vol. 1;.
13. Li, K.; Wu, M. *Effective Software Test Automation: Developing an Automated Software Testing Tool*; John Wiley & Sons, 2006;
14. Ali, N. bin; 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*, doi:10.1007/s10664-018-9670-1.
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. *Applied Sciences (Switzerland)* **2023**, *13*, doi:10.3390/app13159010.
16. Leicht, N.; Blohm, I.; Leimeister, J.M. Leveraging the Power of the Crowd for Software Testing. *IEEE Softw* **2017**, *34*, doi:10.1109/MS.2017.37.
17. Lachmann, R. 12.4 - Machine Learning-Driven Test Case Prioritization Approaches for Black-Box Software Testing.; 2020.
18. Rätzmann, M.; De Young, C. *Software Testing and Internationalization*; Lemoine International, Incorporated, 2003;
19. Broekman, B.; Notenboom, E. *Testing Embedded Software*; Pearson Education, 2003;
20. Elish, K.O.; Elish, M.O. Predicting Defect-Prone Software Modules Using Support Vector Machines. *Journal of Systems and Software* **2008**, *81*, doi:10.1016/j.jss.2007.07.040.
21. Li, J.; He, P.; Zhu, J.; Lyu, M.R. Software Defect Prediction via Convolutional Neural Network. In Proceedings of the Proceedings - 2017 IEEE International Conference on Software Quality, Reliability and Security, QRS 2017; 2017.
22. Ordonez, M.J.; Haddad, H.M. The State of Metrics in Software Industry. In Proceedings of the Proceedings - International Conference on Information Technology: New Generations, ITNG 2008; 2008.
23. Shiva, S.G.; Shala, L.A. Software Reuse: Research and Practice. In Proceedings of the Proceedings - International Conference on Information Technology-New Generations, ITNG 2007; 2007.
24. Zhang, H.; Zhang, X.; Gu, M. Predicting Defective Software Components from Code Complexity Measures. In Proceedings of the Proceedings - 13th Pacific Rim International Symposium on Dependable Computing, PRDC 2007; 2007.
25. Galinac Grbac, T.; Runeson, P.; Huljenić, D. A Second Replicated Quantitative Analysis of Fault Distributions in Complex Software Systems. *IEEE Transactions on Software Engineering* **2013**, *39*, doi:10.1109/TSE.2012.46.
26. Mauša, G.; Galinac Grbac, T. Co-Evolutionary Multi-Population Genetic Programming for Classification in Software Defect Prediction: An Empirical Case Study. *Applied Soft Computing Journal* **2017**, *55*, doi:10.1016/j.asoc.2017.01.050.
27. 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 Proceedings of the Advances in Intelligent Systems and Computing; 2018; Vol. 738.
28. Alqarni, A.; Aljamaan, H. Leveraging Ensemble Learning with Generative Adversarial Networks for Imbalanced Software Defects Prediction. *Applied Sciences* **2023**, *13*, doi:10.3390/app132413319.
29. Babatunde, A.N.; Ogundokun, R.O.; Adeoye, L.B.; Misra, S. Software Defect Prediction Using Dagging Meta-Learner-Based Classifiers. *Mathematics* **2023**, *11*, doi:10.3390/math11122714.
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 (Basel)* **2020**, *12*, doi:10.3390/sym12071147.
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 (Switzerland)* **2021**, *10*, doi:10.3390/electronics10020179.

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. *Applied Sciences (Switzerland)* **2019**, *9*, doi:10.3390/app9132764.
33. Khurma, R.A.; Alsawalqah, H.; Aljarah, I.; Elaziz, M.A.; Damaševičius, R. An Enhanced Evolutionary Software Defect Prediction Method Using Island Moth Flame Optimization. *Mathematics* **2021**, *9*, doi:10.3390/math9151722.
34. Pan, C.; Lu, M.; Xu, B.; Gao, H. An Improved CNN Model for Within-Project Software Defect Prediction. *Applied Sciences (Switzerland)* **2019**, *9*, doi:10.3390/app9102138.
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 (Switzerland)* **2022**, *11*, doi:10.3390/electronics11172707.
36. Tong, H.; Wang, S.; Li, G. Credibility Based Imbalance Boosting Method for Software Defect Proneness Prediction. *Applied Sciences (Switzerland)* **2020**, *10*, doi:10.3390/app10228059.
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. *Applied Sciences (Switzerland)* **2020**, *10*, doi:10.3390/app10238324.
38. Shepperd, M.; Song, Q.; Sun, Z.; Mair, C. Data Quality: Some Comments on the NASA Software Defect Datasets. *IEEE Transactions on Software Engineering* **2013**, *39*, doi:10.1109/TSE.2013.11.
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; 2011; Vol. 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 Proceedings - 18th IEEE International Conference on Machine Learning and Applications, ICMLA 2019; 2019.
41. Freiesleben, T.; Grote, T. Beyond Generalization: A Theory of Robustness in Machine Learning. *Synthese* **2023**, *202*, doi:10.1007/s11229-023-04334-9.
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*, doi:10.1148/ryai.220028.
43. Verkerken, M.; D'hooge, L.; Wauters, T.; Volckaert, B.; De Turck, F. Towards Model Generalization for Intrusion Detection: Unsupervised Machine Learning Techniques. *Journal of Network and Systems Management* **2022**, *30*, doi:10.1007/s10922-021-09615-7.
44. Abdolrasol, M.G.M.; Suhail Hussain, S.M.; 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 (Switzerland)* 2021, *10*.
45. Panerati, J.; Schnellmann, M.A.; Patience, C.; Beltrame, G.; Patience, G.S. Experimental Methods in Chemical Engineering: Artificial Neural Networks–ANNs. *Canadian Journal of Chemical Engineering* 2019, *97*.
46. 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 and AI* **2020**, *2*, doi:10.1016/j.egyai.2020.100028.
47. Zhou, T.; Wang, F.; Yang, Z. Comparative Analysis of ANN and SVM Models Combined with Wavelet Preprocess for Groundwater Depth Prediction. *Water (Switzerland)* **2017**, *9*, doi:10.3390/w9100781.
48. 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*, doi:10.1016/j.compchemeng.2020.106970.
49. 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 Studies in Thermal Engineering* **2021**, *28*, doi:10.1016/j.csite.2021.101406.
50. Lin, C.J.; Wu, N.J. An Ann Model for Predicting the Compressive Strength of Concrete. *Applied Sciences (Switzerland)* **2021**, *11*, doi:10.3390/app11093798.
51. Adadi, A.; Berrada, M. Peeking Inside the Black-Box: A Survey on Explainable Artificial Intelligence (XAI). *IEEE Access* **2018**, *6*, doi:10.1109/ACCESS.2018.2870052.
52. Razavi, S. Deep Learning, Explained: Fundamentals, Explainability, and Bridgeability to Process-Based Modelling. *Environmental Modelling and Software* **2021**, *144*, doi:10.1016/j.envsoft.2021.105159.

53. Chapman, M.; Callis, P.; Menzies, T. JM1/Software Defect Prediction 2004.
54. Gaffney Jr, J.E. Metrics in Software Quality Assurance. In Proceedings of the Proceedings of the ACM'81 conference; 1981; pp. 126–130.
55. Halstead, M.H. Toward a Theoretical Basis for Estimating Programming Effort. In Proceedings of the Proceedings of the 1975 annual conference; 1975; pp. 222–224.
56. McCabe, T.J.; Butler, C.W. Design Complexity Measurement and Testing. *Commun ACM* **1989**, *32*, 1415–1425.
57. McCabe, T.J. A Complexity Measure. *IEEE Transactions on software Engineering* **1976**, 308–320.
58. 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 Evaluation & Assessment in Software Engineering (EASE 2011), 15th Annual Conference on; 2011; pp. 96–103.
59. Wilson, D.L. Asymptotic Properties of Nearest Neighbor Rules Using Edited Data. *IEEE Trans Syst Man Cybern* **1972**, *2*, doi:10.1109/TSMC.1972.4309137.
60. Lewis, D.D.; Catlett, J. Heterogeneous Uncertainty Sampling for Supervised Learning. In Proceedings of the Proceedings of the 11th International Conference on Machine Learning, ICML 1994; 1994.
61. Kubat, M.; Matwin, S. Addressing the Curse of Imbalanced Training Sets: One-Sided Selectio. In Proceedings of the International Conference on Machine Learning; 1997.
62. Chambers, R.L. Robust Case-Weighting for Multipurpose Establishment Surveys. *JOURNAL OF OFFICIAL STATISTICS-STOCKHOLM-* **1996**, *12*, 3–32.
63. Chawla, N. V.; Bowyer, K.W.; Hall, L.O.; Kegelmeyer, W.P. SMOTE: Synthetic Minority over-Sampling Technique. *Journal of Artificial Intelligence Research* **2002**, *16*, doi:10.1613/jair.953.
64. Breiman, L. Random Forests. *Mach Learn* **2001**, *45*, doi:10.1023/A:1010933404324.
65. 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*, doi:10.1016/j.oregeorev.2015.01.001.
66. Liaw, A.; Wiener, M. Classification and Regression by RandomForest. *R News* **2002**, *2*.
67. Marchese Robinson, R.L.; 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*, doi:10.1021/acs.jcim.6b00753.
68. Kumaravel, A.; Vijayan, T. Comparing Cost Sensitive Classifiers by the False-Positive to False- Negative Ratio in Diagnostic Studies. *Expert Syst Appl* **2023**, *227*, doi:10.1016/j.eswa.2023.120303.
69. 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 Proceedings of Machine Learning Research; 2018; Vol. 88.
70. Stone, M. Cross-Validatory Choice and Assessment of Statistical Predictions. *Journal of the Royal Statistical Society: Series B (Methodological)* **1974**, *36*, doi:10.1111/j.2517-6161.1974.tb00994.x.
71. 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; 1995; Vol. 2.
72. Menzies, T.; Di Stefano, J.S. How Good Is Your Blind Spot Sampling Policy. In Proceedings of the High Assurance Systems Engineering, 2004. Proceedings. Eighth IEEE International Symposium on; 2004; pp. 129–138.
73. Seliya, N.; Khoshgoftaar, T.M.; Van Hulse, J. A Study on the Relationships of Classifier Performance Metrics. In Proceedings of the Proceedings - International Conference on Tools with Artificial Intelligence, ICTAI; 2009.
74. Menzies, T.; Di Stefano, J.S. How Good Is Your Blind Spot Sampling Policy? In Proceedings of the Proceedings of IEEE International Symposium on High Assurance Systems Engineering; 2004; Vol. 8.