applied sciences

MDPI

Article

Software Defect Prediction Using Stacking Generalization of Optimized Tree-Based Ensembles

Amal Alazba [1,2,†] and Hamoud Aljamaan [1,*,†]

[1] Information and Computer Science Department, King Fahd University of Petroleum and Minerals, Dhahran 31261, Saudi Arabia
[2] Department of Information Systems, King Saud University, Riyadh 11362, Saudi Arabia; aalazba@ksu.edu.sa
* Correspondence: hjamaan@kfupm.edu.sa
† These authors contributed equally to this work.

Abstract: Software defect prediction refers to the automatic identification of defective parts of software through machine learning techniques. Ensemble learning has exhibited excellent prediction outcomes in comparison with individual classifiers. However, most of the previous work utilized ensemble models in the context of software defect prediction with the default hyperparameter values, which are considered suboptimal. In this paper, we investigate the applicability of a stacking ensemble built with fine-tuned tree-based ensembles for defect prediction. We used grid search to optimize the hyperparameters of seven tree-based ensembles: random forest, extra trees, AdaBoost, gradient boosting, histogram-based gradient boosting, XGBoost and CatBoost. Then, a stacking ensemble was built utilizing the fine-tuned tree-based ensembles. The ensembles were evaluated using 21 publicly available defect datasets. Empirical results showed large impacts of hyperparameter optimization on extra trees and random forest ensembles. Moreover, our results demonstrated the superiority of the stacking ensemble over all fine-tuned tree-based ensembles.

Keywords: machine learning; ensemble learning; software defect; hyperparameter optimization; stacking generalization

check for updates

Citation: Alazba, A.; Aljamaan, H. Software Defect Prediction Using Stacking Generalization of Optimized Tree-Based Ensembles. *Appl. Sci.* **2022**, *12*, 4577. https://doi.org/10.3390/app12094577

Academic Editor: Paolino Di Felice

Received: 4 March 2022
Accepted: 27 April 2022
Published: 30 April 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

It is important to produce high-quality software, yet it is difficult to achieve such quality because of time and budget constraints. Identifying and correcting software defects using the most efficient quality assurance techniques is a challenging and expensive process [1]. Software defect prediction may be utilized to classify faulty fragments of software systems. Therefore, quality assurance methods can be implemented on the selected parts instead of a complete system inspection. The process of finding and classifying software components that are vulnerable to defects through machine learning techniques is known as software defect prediction. There are several benefits to software defect prediction—for instance, decreasing the expensiveness of testing and producing software products in a timely manner, thereby enhancing the overall quality of software products.

Various machine learning classifiers, such as decision trees and ensemble learning, have recently been employed to improve the detection of software defects [2]. A variety of software metrics derived from the source code are typically used to construct the predictive models. McCabe metrics, Halstead metrics, and static code metrics are the most often used for defect prediction [2]. Ensemble learning, such as stacking and boosting, was shown to be successful in predicting software defects, and it exhibited excellent prediction outcomes in comparison with individual classifiers [3]. However, most of the previous work has utilized ensembles models in the context of software defect prediction with the default hyperparameter values, which are considered suboptimal [4]. Hyperparameter optimization has a large impact on the performances of classifiers, and the few studies that have utilized an optimization technique reported improved prediction performance [5,6].

In this study, we investigated the effects of stacked ensembles and hyperparameter optimization on tree-based ensembles. We predicted software defects using tuned and untuned tree-based ensembles based on a set of software metrics. The included ensembles were: extra trees, XGBoost, CatBoost, gradient boosting and histogram gradient boosting. The selection of those ensembles was based on the fact that very few studies have explored these ensembles. Furthermore, we explored the prediction performance of a stacking ensemble built using fine-tuned tree-based ensembles. We conducted a comprehensive empirical study to examine the efficacy of various ensemble models at predicting software defects.

The rest of the paper is structured in the following manner. The background on tree-based and stacking ensembles used to predict software defects is addressed in Section 2. Section 3 examines previous work in ensemble learning and hyperparameter optimization for software defect prediction. Section 4 explains the process of the empirical study that was performed. The findings and discussion are presented in Section 5. Finally, Section 6 summarizes the conclusions and possible future work.

2. Background

In this study, we utilized various tree-based boosting and bagging homogeneous ensembles and the stacking generalization heterogeneous ensemble for predicting software defect. An overview of the ensemble models used is discussed in this section.

2.1. Tree-Based Ensembles

Ensemble learning [7] is a machine learning technique that creates a final prediction model by combining many machine learning classifiers. Ensembles can learn complex data patterns for decision making and class prediction. In software defect prediction, there are two primary types of problem: regression in which the classifier predicts the number of defects, and classification in which the classifier predicts the label of an instance (i.e., defective or non-defective). Ensemble learning has two phases [7]: (1) training the base classifiers which consist of multiple individual classifiers and (2) construct a final prediction model using averaging or voting that are computed from the results of the base classifiers [8]. Ensembles are known to be homogeneous if only one type of algorithm is used in the base classifiers and are considered heterogeneous if various types of algorithm are used as the base classifiers.

The main focus of this study is tree-based ensembles, which are boosting and bagging homogeneous ensembles where the base classifier is a decision tree [9]. Boosting [10] is a sequential ensemble method that combines multiple classification methods to create a low-bias model and thus promote better prediction performance. The key idea behind boosting is to construct classifiers iteratively and sequentially, with each model relying on the previous model, taking into account misclassified instances that were predicted by the previous classifiers. There are many boosting ensembles, the key variations being in how the base classifies are trained and combined. Bagging [11] or bootstrap aggregation is a parallel ensemble method that combines several models to produce a model with low variance and hence improved prediction performance. The essence behind bagging is to train multiple base classifiers independently and then combine the outputs by averaging or voting. Bagging's main advantage is the ability to parallelize the process of training the base classifiers, since they are fitted independently. The following are brief descriptions of the tree-based ensembles that were chosen:

- **AdaBoost (Ada)** [10], or adaptive boosting, is a well-known boosting algorithm introduced by Freund and Schapire. AdaBoost fits a series of base classifiers and then changes the weights of the instances by giving the misclassified ones a higher weight and then fitting the updated weights with a new base learner. The final prediction is calculated by integrating the results from all base classifiers using a weighted majority vote approach, in which each base classifier contributes based on its results (i.e., assigned a greater weight).

- **Random forest (RF)** [12] is a bagging algorithm that employs a large number of small decision trees, each of which is developed from random dataset subsets. To increase the tree's diversity, a random subset of features is chosen at each node to produce the best split. Overfitting is less likely due to the randomness of the dataset and features. The majority vote is used in a classification problem to determine the final class label.
- **Extra trees (ET)** [13], or extremely randomized trees, is similar to the RF algorithm but with additional randomness. ET differs from RF in two ways: (1) each decision tree is constructed using the entire dataset, and (2) it randomly selects the splits at each node (i.e., does not select the best splits).
- **Gradient boosting (GB)** [14] is a generalization of the AdaBoost ensemble that can be constructed utilizing a variety of loss functions. GB, unlike AdaBoost, fits a new base classifier using gradients rather than the weight of misclassified instances. Using GB will boost the efficiency of fitting the base classifiers, but memory usage and processing time are inefficient.
- **Histogram-based gradient boosting (HGB)** [15], or histogram-based gradient boosting, is a boosting ensemble that selects the best splits easily and reliably using feature histograms. It is more efficient than GB in terms of processing speed and memory utilization.
- **XGBoost (XGB)** [16], or extreme gradient boosting, is similar to the GB algorithm, but instead of gradients, it fits a new base classifier using second-order derivatives of the loss function. XGB is thought to be more precise and effective than GB.
- **CatBoost (CAT)** [17], or categorical boosting, is a boosting ensemble with two key characteristics: (1) it handles categorical features with one-hot encoding, and (2) it produces oblivious decision trees as base classifiers with the same features used as a splitting criterion for all nodes within the same tree level. Oblivious trees are symmetrical; thus, overfitting is minimized and training time is reduced.

2.2. Stacking Ensemble

Stacking [18] is a heterogeneous ensemble model that produces the final prediction model by combining several base classifiers through a meta-classifier. Different learning algorithms are used by the base classifiers and are trained using the whole training dataset. To create the final prediction model, the results of the base classifiers are fed into the meta-classifier as a training set. To resolve any overfitting, the training dataset needs to be split into two parts, one to train the base classifiers and one for the meta-classifier. The pseudo-code in Algorithm 1 summarizes the algorithm for the stacking ensemble [19]. The algorithm takes as input a training data set D, which consists of m instances, and each instance i has (x, y), where x represents a feature vector and y is the class label. It returns a stacking ensemble model H after three main steps:

- Step 1—first-level classifiers: Assume that we have T base classifier. For each base classifier $t \in T$ learn and fit the classifier using $\mathcal{D}$.
- Step 2—construct new datasets: The outputs of the T base classifiers are used as input to the meta-classifier. To create new datasets, for each instance (x, y), construct a new instance (x', y) such that (1) x' is a vector of length T and $x' = \{h_1(x), h_2(x), \ldots, h_T(x)\}$, where $h_t(x)$ is the prediction output of a base classifier t and (2) y represents the original class labels.
- Step 3—a second-level classifier: Using the dataset created in step 2, fit and teach a meta-classifier which produces a final prediction by combining the outputs of all base classifiers.

Algorithm 1: Stacking ensemble.

Require: Training data $\mathcal{D} = \{\mathbf{x}_i, \mathrm{y}_i\}_{i=1\ldots m}(\mathbf{x}_i \in \mathrm{R}^{\mathrm{n}}, \mathrm{y}_i \in \mathrm{Y})$
Ensure: An ensemble classifier H
1: Step 1: Learn first-level classifiers
2: **for** $t \leftarrow 1$ to T **do**
3: Learn a Tree-based ensemble h_t based on $\mathcal{D}$
4: **end for**
5: Step 2: Construct new data sets from $\mathcal{D}$
6: **for** $i \leftarrow 1$ to m **do**
7: Construct a new data set that contains $\{\mathbf{x}'_i, \mathbf{y}_i\}$, where $\mathbf{x}'_i = \{h_1(\mathbf{x}_i), h_2(\mathbf{x}_i), \ldots, h_T(\mathbf{x}_i)\}$
8: **end for**
9: Step 3: Learn a second-level classifier
10: Learn a new classifier h' based on the newly constructed data set
11: **return** $H(\mathbf{x}) = h'(h_1(\mathbf{x}), h_2(\mathbf{x}), \ldots, h_T(\mathbf{x}))$

3. Literature Review

Machine learning-based software defect prediction is a hot topic in academia [2], and various machine learning models have been investigated. Ensemble learning has recently gotten a lot of interest in the domain of software defect prediction. Although machine learning techniques have been used in software defect prediction, most studies have utilized the default hyperparameters for these techniques which are suboptimal. In this section, we discuss research that has used ensemble learning to detect software defects. Then, we review studies that have utilized hyperparameter optimization to enhance the detection of software defects.

3.1. Ensemble Learning

Ensemble learning is the process by which several machine learning classifiers are combined to improve detection performance [7,8]. Ensemble learning has proven to be an effective technique [20]. A wide variety of ensemble learning models have been investigated in the literature for software defect prediction, which are summarized in Table 1.

Ensemble learning can be classified into two categories: homogeneous [20,21] and heterogeneous [22,23] ensembles. Aljamaan and Elish [20] explored the detection performance of bagging and boosting homogeneous ensembles using the KC1 NASA dataset. They examined the two ensembles' prediction performances compared to six individual classifiers and concluded that bagging and boosting ensembles performed better than individual classifiers. A similar study by Yohannese et al. [21] investigated the prediction performances of bagging and boosting ensembles on eight NASA software defect datasets. They utilized feature selection (i.e., information gain IG) to reduce the dimensionality, along with SMOTE, which is an oversampling technique to balance the defective and non-defective instances. Experimental results indicated that using ensembles along with IG and SMOTE techniques can increase the detection performance.

Furthermore, heterogeneous ensembles have been explored to detect software defects. In one study led by Pandey et al. [23], the effect of the heterogeneous voting ensemble on the prediction performance of software defects was studied. They combined different classifiers to build the voting ensemble. Then, 12 NASA datasets were used to assess the classifier. The voting ensemble outperformed individual classifiers. In Petrić et al. [22], a different type of heterogeneous ensemble, the stacking ensemble, was explored. They combined multiple base classifiers and then assessed the proposed ensemble on eight datasets. They concluded that compared to bagging, the stacking ensemble with diversity selection performed better.

The comparison between homogeneous and heterogeneous ensembles has been also explored in software defect prediction. In Hussain et al. [24], homogeneous boosting

ensembles were compared to heterogeneous voting and stacking ensembles in terms of prediction performance. They evaluated the performances of the ensembles using 12 defects datasets. Their experimental results show that stacking ensembles have better detection performances compared to boosting and voting ensembles. In a study led by Li et al. [25], the voting heterogeneous ensemble was compared with four homogeneous ensembles: RF, Ada, random subspace method (RSM) and bagging. They used five NASA software defect datasets to evaluate the ensembles and utilized SMOTE to resolve the data imbalance issue. Their results showed that RF and voting outperformed other ensembles in detecting defects.

Other empirical studies used average probability ensemble learning techniques for combining the base classifiers. Tran et al. [26] and Tong et al. [27] constructed ensembles with two-stages utilizing three homogeneous ensembles: RF, Ada and bagging. These ensembles were combined using the weighted average probabilities; then, they assessed the proposed ensembles on 12 NASA MDP defects datasets. The results of [27] revealed that the proposed ensemble has a superior performance over other ensembles. In [26], they used a feature selection method (i.e., greedy forward selection) and they reported a superior detection performance. The average probability ensemble and GFS have also been used by Laradji et al. [28]. In their study, seven base classifiers were utilized to construct the ensemble model. They evaluated the proposed ensemble against W-SVMs and RF on six publicly available defects datasets. Their empirical study revealed the superiority of the proposed ensemble over the other two classifiers.

Table 1. Comparison between prior studies of ensembles in software defect prediction.

Ref	Dataset	Base Classifiers	Ensembles	Statistical Analysis
[20]	KC1 (class)	MLP, RBF, BBN, NB, SVM, DT	Bagging, AdaBoost	no
[21]	ar1, ar4, JM1′, KC2, MC1″, MW1′, PC3′, PC4″	J48	Bagging, AdaBoost.M2	no
[23]	CM1, KC1, KC2, KC3, MC1, MC2, PC1, JM1, MW1, PC2, PC3, PC4	IBK, MLP, SVM, RF, NB, Logistic Boost, PART, JRip, J48, Decision Stump	Voting	Wilcoxon
[22]	ant-1.5, ant-1.6, ant-1.7, jedit-4.1, jedit-4.2, tomcat. xalan-2.5, xalan-2.6	NB, DT, KNN, SMO	Stacking	Wilcoxon
[24]	Ant-1.7, Camel-1.6, e-learning, Forrest-0.8, Jedit-4.3, Tomcat, Xalan-2.7, Xerces-1.4, Zuzel, Berek, Pbean2, Velocity-1.6	NB, LR, J48, VP, SMO	AdaBoostM1, Voting, Stacking	no
[25]	CM1, JM1, KC1, KC2, PC1	J48	AdaBoost, Bagging, RSM, RF, Voting	no
[27]	CM1, KC1, KC2, KC3, MC1, MC2, MW1, PC1, PC2, PC3, PC4, JM1	Bagging, RF, AdaBoost	Weighted average probabilities	Wilcoxon
[26]	KC1, KC2, KC3, PC1, PC2, PC3, PC4, MC1, MC2, CM1, JM1, MW1	Bagging, RF, AdaBoost	Weighted average probabilities	no
[28]	Ant-1.7, Camel-1.6, KC3, MC1, PC2, PC4	RF, GB, SGD, W-SVMs, LR, M-NB, B-NB	Average probability	no
This work	21 Defect Datasets	Ada, RF, ET, GB, HGB, XGB, CAT	Stacking	Wilcoxon

3.2. Hyperparameter Optimization

Machine learning techniques have been used in software defect prediction; however, most studies have utilized the default hyperparameters for these techniques, which are suboptimal. Nevertheless, the effects of hyperparameter tuning have been investigated in a few studies, and it has been shown to enhance the prediction performances of the classifiers [5,29,30]. Table 2 summaries relevant papers that have used hyperparameter optimization for software defect prediction. Khan et al. [29] utilized an optimized artificial

immune network (opt-aiNet) to optimize the hyperparameters of KNN, SVM, NB, DT, LDA, RF and AdaBoost classifiers. Five Eclipse defect datasets were used, including JDT Core, PDE UI, Equinox, Mylyn and Lucene. Their results showed that the classifiers with optimized hyperparameters performed better than the default ones.

Tantithamthavorn et al. [5] explored the effect of hyperparameter optimization using grid search on the prediction performances of 26 classifiers for software defect prediction. The classifiers belonged to 11 classification families, including NB, KNN, LR, NN, partial least squares (PLS), discrimination analysis (DA), rule-based, DT, SVM, bagging and boosting. A total of 18 datasets from NASA, Proprietary, Apache and Eclipse were used in their experiment. Their results revealed that the hyperparameter optimization had a great impact on the classifiers' performances. They extended their work [30] by investigating three more optimization techniques: random search, the genetic algorithm and differential evolution. They compared the four optimization methods on only four classifiers. They concluded that the four optimization methods yielded similar results in terms of increasing the prediction performances of the classifiers.

Osman et al. [6] used grid search to optimize the hyperparameters of KNN and SVM for software defect prediction. They utilized five Eclipse software defect datasets: JDT Core, PDE UI, Equinox, Mylyn and Lucene. Their results showed increases in the accuracy of KNN and SVM defect prediction models by 20% and 10%, respectively. A differential evolution algorithm has been used by Fu et al. [31] to optimize three classifiers: Where-based Learner, DT and RF. They utilized 17 software defect datasets. They concluded that hyperparameter optimization is simple and yields very good results compared with the default values. Öztürk et al. [32] investigated the use of grid search and gradient boosting machine for hyperparameter optimization of RF and SVM classifiers. They used 20 software defect datasets to explore the impacts of tuning in cross-project defect prediction (CPDP) and within-project defect prediction (WPDP). Their overall results showed that the prediction performance improved when hyperparameter optimization was used.

Table 2. Comparison between prior studies that used hyperparameter optimization.

Ref	Datasets	Optimization Methods	Classifiers	Ensembles
[29]	Eclipse JDT Core, Eclipse PDE UI, Equinox, Mylyn, Lucene	opt-aiNet	KNN, SVM, NB, DT, LDA, RF, AdaBoost	Yes
[5]	JM11, PC51, Prop-12, Prop-22, Prop-32, Prop-42, Prop-52, Camel 1.22, Xalan 2.52, Xalan 2.62, Platform 2.03, Platform 2.13, Platform 3.03, Debug 3.44, SWT 3.44, JDT5, Mylyn5, PDE5	Grid search	NB, KNN, LR, NN, PLS, DA, Rule-based, DT, SVM, Bagging, Boosting	Yes
[30]	JM11, PC51, Prop-12, Prop-22, Prop-32, Prop-42, Prop-52, Camel 1.22, Xalan 2.52, Xalan 2.62, Platform 2.03, Platform 2.13, Platform 3.03, Debug 3.44, SWT 3.44, JDT5, Mylyn5, PDE5	Grid search, Random search, Genetic algorithm, Differential evolution	NB, KNN, LR, NN, PLS, DA, Rule-based, DT, SVM, Bagging, Boosting	Yes
[6]	Eclipse JDT Core, Eclipse PDE UI, Equinox, Mylyn, Lucene	Grid Search	KNN, SVM	No
[31]	antV0, antV1, antV2, camelV0, camelV1, ivy, jeditV0, jeditV1, jeditV2, log4j, lucene, poiV0, poiV1, synapse, velocity, xercesV0, xercesV1	Differential evolution	Where-based Learner, DT, RF	Yes
[32]	ar1, ar3, ar4, ar5, ar6, cm1, jm1, kc1, kc2, kc3, pc1, pc2, pc3, pc4, pc5, Eclipse JDT Core, Eclipse PDE UI, Equinox framework, Lucene, Mylyn	Grid Search	RF, SVM	No
This work	21 Defect Datasets	Grid Search	Ada, RF, ET, GB, HGB, XGB, CAT	Yes

3.3. Summary

Machine learning techniques have proved to be effective for predicting software defects. Ensemble models showed promising prediction results over individual classifiers. The most commonly investigated ensembles are bagging, boosting and voting. Despite the fact that the stacking ensemble has showed superior performance over other homogenous and heterogenous ensembles [24], only two studies have investigated the use of the stacking ensemble in software defect prediction. Moreover, most of machine learning classifiers that have been investigated in the context of software defect prediction have utilized classifiers with default hyperparameter values, which are considered suboptimal [4]. Hyperparameter optimization has a large impact on the performances of the classifiers, and the few studies that have utilized an optimization technique reported increases in prediction performance. In this study, we investigated the effects of hyperparameter optimization on tree-based ensembles. Additionally, we explored the prediction performance of a stacking ensemble that was built using fine-tuned tree-based ensembles.

4. Empirical Study

This section discusses the details of our empirical study. In this study, we explore the capabilities of tuned tree-based ensembles and stacking ensembles for software defect prediction. We used Python to implement the empirical study, i.e., for tuning, training and testing all ensembles.

4.1. Goal

Software defect prediction using machine learning approaches has been investigated in previous studies with a focus on using the default hyperparameter configuration. However, recent studies have reported the impact of hyperparameter optimization on the performances of the classifiers. Tantithamthavorn et al. [5] have pointed out the need for exploring hyperparameter optimization in ensemble learning. The main objective of our experiment was to investigate to what extent hyperparameter optimization would impact the prediction performances of ensemble models.

Using the Goal–Question–Metric (GQM) template [33], we defined the goal of this experiment as: **evaluating** fine-tuned tree-based ensembles and stacking ensemble models of fine-tuned tree-based ensembles made for the **purpose** of software defect prediction, with **respect** to their detection performance measures (F-measure, AUC and Brier), and comparing them with tree-based ensembles with default hyperparameters from the **perspectives** of researchers and practitioners within the **context** of 21 software defect datasets obtained from different domains. The following research questions were formulated to attain our objectives:

- **RQ 1.** To what extent does hyperparameter optimization increase the prediction performance of an ensemble?
- **RQ 2.** To what extent does stacking generalization of fine-tuned tree-based ensembles increase defect prediction?

An overview of the steps to conduct the experiment is shown in Figure 1. First, we selected defect datasets. Then, we used an optimization method to optimize the hyperparameters of the tree-based ensembles. Next, we constructed the ensemble models (tree-based and stacking ensembles). After that, we validated each model and measured the prediction performance. Finally, we estimated the effect size and ranked the ensembles based on their performances. The following subsections describe each step.

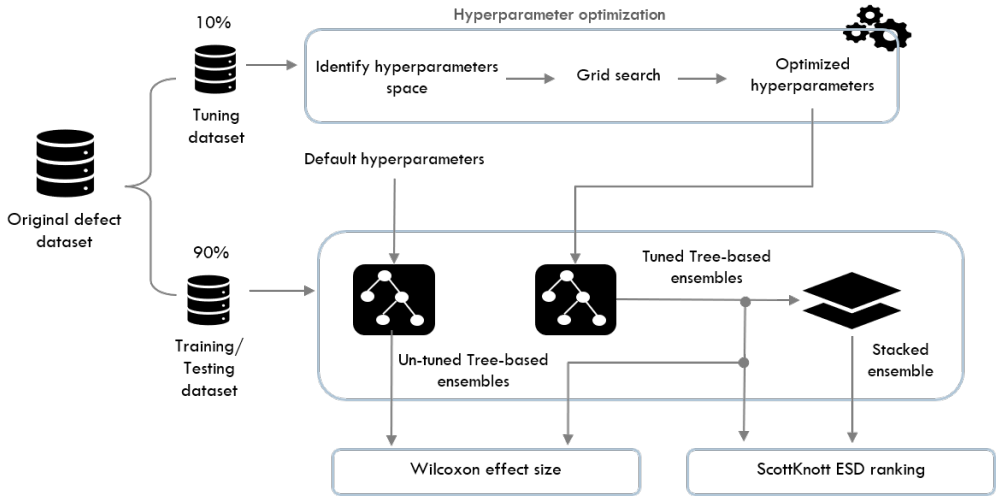

Figure 1. An overview of the conducted empirical study.

4.2. Datasets

In this empirical investigation, 102 software defect datasets were utilized. These datasets were publicly available and provided by Shepperd et al. [34], D'Ambros et al. [35], Jureczko et al. [36], Wu et al. [37], Zimmermann et al. [38], Kim et al. [39] and Turhan et al. [40]. For each dataset, we calculated the number of events per variable (EPV) [41] and the defective rate. EPV is the ratio of the number of defective instances to the number of independent variables. Previous study has shown the effect of EPV on defect prediction models: higher EPV values produce more stable prediction models [42]. Peduzzi et al. [41] reported that roughly 10 EPV were essential for accurate estimation of machine learning models. Therefore, among the 102 datasets, we selected 25 datasets that had EPVs greater than 10. Moreover, we calculated the defective rate for each dataset, and we excluded the datasets with defective rates greater than 50%.

The final set of datasets used in this paper is listed in Table 3. A total of 21 publicly available software defect datasets were used. To ensure variety and reduce bias, the selected datasets were collected by different research groups and from various domains. Each dataset was of a different size (i.e., number of instances) and granularity (i.e., instances represent a method, class or file). Each instance consisted of: (1) a single binary value indicating whether the instance was defective or non-defective (i.e., dependent variable) and (2) a set of software metrics (i.e., independent variables).

4.3. Hyperparameter Optimization

Hyperparameter optimization or tuning refers to the process of searching for and finding the optimal hyperparameters values of machine learning models. There are different hyperparameter optimization techniques, such as grid search, random search, genetic algorithm and differential evolution. Previous research [30] has shown that the four techniques (i.e., grid search, random search, genetic algorithm and differential evolution) yield similar prediction performances. However, this recommendation was limited to the techniques selected in that paper, and might not be applicable for our selected tree-based ensembles. Optimizing tree-based ensembles had not been done previously.

Grid search [43] is a hyperparameter optimization technique where all possible combinations of hyperparameters values in the search space are explored. The search space consists of a set of hyperparameters being explored and a set of possible values for each hyperparameter. For each combination, the grid search algorithm builds and evaluates a model and the hyperparameters values of the best performing model are returned as optimal values. In this experiment, we used the grid search algorithm since we had a reasonable search space, and we wanted to conduct a controlled experiment where the hyperparameter values being explored were consistent across all datasets.

Table 3. Software defect datasets used in the experiment.

Authors	Dataset Name	Ind. Variables	Defective	Non-Defective	Instances	EPV	Defective %	Granularity
Shepperd et al. [34]	JM1	21	1672	6110	7782	79.6	21%	Method
	KC1	21	314	869	1183	15.0	27%	Method
	PC5	38	471	1240	1711	12.4	28%	Method
D'Ambros et al. [35]	Eclipse JDT Core	15	206	791	997	13.7	21%	Class
	Eclipse PDE UI	15	209	1288	1497	13.9	14%	Class
	Mylyn	15	245	1617	1862	16.3	13%	Class
Jureczko et al. [36]	camel 1.2	20	216	549	765	10.8	28%	Class
	lucene 2.4	20	203	333	536	10.2	38%	Class
	prop-1	20	2436	20,622	23,058	121.8	11%	Class
	prop-2	20	1514	11,645	13,159	75.7	12%	Class
	prop-3	20	840	8027	8867	42.0	9%	Class
	prop-4	20	1299	7274	8573	65.0	15%	Class
	prop-43	20	341	11,338	11,679	17.1	3%	Class
	prop-5	20	2720	17,686	20,406	136.0	13%	Class
	xalan 2.5.0	20	387	558	945	19.4	41%	Class
	xalan 2.6.0	20	411	759	1170	20.6	35%	Class
Zimmermann et al. [38]	Eclipse 2.0	31	2610	4119	6729	84.2	39%	File
	Eclipse 2.1	31	2139	5749	7888	69.0	27%	File
	Eclipse 3.0	31	2913	7680	10,593	94.0	27%	File
Kim et al. [39]	SWT	17	653	832	1485	38.4	44%	File
	Debug	17	263	802	1065	15.5	25%	File

In this study, we used the grid search to find the optimal hyperparameters of ensemble models. First, we identified the search space (i.e., hyperparameters and the possible values) for each ensemble model shown in Table 4. Then, we utilized the tuning dataset (the original dataset was divided into two, 10% for tuning and 90% for training and testing) to find the optimal values of the hyperparameters of each ensemble model using grid search and the search space. For each ensemble model, all possible hyperparameter combinations were explored by building and evaluating multiple models using stratified 2-fold cross-validation with five repeats [44]. After that, the optimal hyperparameters values of the best performing model were identified based on the AUC scores. Finally, the optimal values were used to build the ensemble models on the training/testing dataset.

4.4. Ensemble Models Construction and Validation

In this experiment, the dataset was partitioned into two groups: 10% of the dataset used in hyperparameter tuning (tuning dataset) and 90% used for training and testing ensemble models (training/testing dataset). Seven tree-based ensemble models, including random forest (RF), extra trees (ET), AdaBoost (Ada), gradient boosting (GB), histogram-based gradient boosting (HGB), XGBoost (XGB) and CatBoost (CAT), and one stacking ensemble model, were constructed. For each tree-based ensemble, two models were trained and tested, one using the optimal hyperparameter values obtained by the optimization algorithm and one using the default hyperparameters. We constructed the stacking ensemble utilizing the tuned tree-based ensemble models as base models and the logistic regression as a meta model. In this study, we used Python to implement our machine learning pipeline using the scikit-learn framework [45].

Table 4. Hyperparameters space investigated in this empirical study.

Ensemble	Hyperparameter Name	Description	Optimized Value Range
Random Forest (RF) Extra Trees (ET)	n_estimators	Number of trees in the forest.	[100, 50, 40, 30]
	max_depth	Maximum depth of the tree.	[1, 4 ,8]
	min_samples_leaf	Minimum number of samples at a leaf node.	RF = [1, 10, 5] ET = [1, 2, 4]
	criterion	Measure the quality of a split.	[gini, entropy]
AdaBoost (Ada)	n_estimators	Number of estimators at which boosting is terminated.	[50, 100, 1000]
	learning_rate	The step size of the loss function.	[1, 0.1, 0.001, 0.01]
Gradient Boosting (GB)	n_estimators	Number of estimators at which boosting is terminated.	[100, 50, 500, 1000]
	learning_rate	The step size of the loss function.	[0.1, 0.001, 0.01]
	min_samples_leaf	Minimum number of samples at a leaf node.	[1, 10, 5]
	max_depth	Maximum depth of individual tree.	[3, 7, 9]
	loss	The loss function to be optimized in the boosting process.	[deviance, exponential]
Hist Gradient Boosting (HGB)	max_iter	Number of iterations of the boosting process.	[100, 50, 500, 1000]
	learning_rate	The step size of the loss function.	[0.1, 0.001, 0.01]
	min_samples_leaf	Minimum number of samples at a leaf node.	[20, 10, 5]
	max_depth	Maximum depth of individual tree.	[None, 1, 3, 5]
	loss	The metric to use in the boosting process.	[auto, binary crossentropy, categorical crossentropy]
XGBoost (XGB)	n_estimators	Number of gradient boosted trees.	[100, 50, 500, 1000]
	learning_rate	The step size of the loss function.	[0.3, 0.1, 0.001, 0.01]
	max_depth	Maximum tree depth for base learners.	[6, 3, 4, 5]
CatBoost (CAT)	n_estimators	Number of estimators at which boosting is terminated.	[1000, 100, 50, 500]
	loss_function	The metric to use in the boosting process.	[Logloss, MultiClass]
	learning_rate	The step size of the loss function.	[0.03, 0.001, 0.01, 0.1]
	depth	Depth of the tree.	[1, 5, 10]
	min_data_in_leaf	Minimum number of samples at a leaf node.	[1, 10]

Prior to building the ensemble model, two pre-processing steps were performed: data transformation and missing value handling:

- Data transformation. Data transformation is recommended to boost the performances of machine learning models [46]. Therefore, we used a logarithmic data transformation where each value of the independent variables was transformed using $x = \log(1 + x)$, where x is the numerical value of the variable. We selected the log transformation because it has been used previously in software defect prediction [30,46].
- Handling missing values. Missing data can be handled in a variety of ways, including mean imputation, deletion and median imputation [47]. For all datasets, we utilized the mean imputation approach to deal with missing values. The mean of all values belonging to the independent variable was used to fill in the missing values. It is one of the most often used approaches and has been proved to yield good results in supervised learning tasks [48].

Given the fact that non-defective cases make up the vast majority of software defect datasets, we did not employ any sampling techniques (e.g., SMOTE) to balance the dataset.

We employed a stratified 10-fold cross-validation method [49], repeated 10 times, to validate the performances of ensemble models. Each dataset was divided into k equal-sized folds, with one-fold serving as a testing dataset for the model's evaluation and the remaining k-1 folds being used for training the model. This technique was performed ten times (i.e., k = 10), with each fold serving as a testing dataset precisely once. To get a final estimate, the detection performance from the k iterations was averaged. Each fold in stratified cross-validation has a proportion of instances belonging to each class that is roughly equal to the proportion in the original dataset. The key benefit of this method is that it uses all instances for both model training and testing. The whole process was repeated 10 times, resulting in 100 independent runs for each investigated model. Finally, the reported model performance was the average of the 100 runs. Repeating the 10 fold cross validation ten times provides a model performance estimate with little bias and low variance [42,50].

4.5. Evaluation Measures

To assess the ensemble models' prediction performances, we chose a set of evaluation metrics. One of the selected measures is a threshold-dependent measure (F-measure), and the other two are threshold-independent measures (AUC and Brier). All of the selected measure were employed by previous studies to assess the performance of defect prediction. The used evaluation measures are:

- F-measurement (F-measure): is the harmonic mean of both precision and recall. F-measurement is calculated as:

$$\text{F-measure} = 2 \times \frac{precision \times recall}{precision + recall} \times 100 \quad (1)$$

$$Precision = \frac{\text{TP}}{\text{FP} + \text{TP}} \quad (2)$$

$$Recall = \frac{\text{TP}}{\text{FN} + \text{TP}} \quad (3)$$

  where true positives (TP) is the number of defective instances correctly classified as defective, false positives (FP) is the number of defective instances incorrectly classified as non-defective and false negatives (FN) is the number of non-defective instances incorrectly classified as defective.
- Area under the curve (AUC): The percentage of the area that is underneath the receiver operator characteristic (ROC) curve. True positive rates are plotted against the false positive rates. The AUC metric calculates the area under the curve, and a large area indicates high performance. AUC can have a value between 0 and 1: 1 indicates the best performance and zero is considered the worst performance.
- Brier [51]: The mean squared difference between the predicted probability and the actual outcome. It can be calculated using the following formula:

$$\text{Brier} = \sum_{i=1}^{n} (p_i - y_i)^2 \quad (4)$$

  where p_i is the predicted probability and y_i is the actual outcome. Brier has a range of values from 0 to 1, with 0 denoting the highest performance, 1 denoting the worst and 0.5 denoting a random estimate.

4.6. Effect Size Estimation

In order to answer RQ1, we used the Wilcoxon effect size test to evaluate the magnitude of hyperparameter optimization. Moreover, we used the Scott–Knott effect size difference (ESD) test to rank the ensemble models. Both estimation techniques are described in the following subsections.

4.6.1. Wilcoxon Effect Size

We used the non-parametric Wilcoxon effect size [52] test to examine the magnitude of the prediction performance between the tuned and untuned tree-based ensembles. We selected a non-parametric effect size test because the data were not normally distributed. The normality distribution was checked using the Shapiro–Wilk test. The Wilcoxon effect size test calculates the correlation coefficients using the Z-score employing; then, the following formula is used to estimate the effect size:

$$r = \frac{Z}{\sqrt{n}} \quad (5)$$

where n is the total number of samples on which Z is based. We used the thresholds proposed by Cohen to categorize the magnitude of r effect size [53]:

$$\text{Effect size} = \begin{cases} \text{negligible} & \text{if } r \leq 0.1 \\ \text{small} & \text{if } 0.1 < r \leq 0.3 \\ \text{medium} & \text{if } 0.3 < r \leq 0.5 \\ \text{large} & \text{if } 0.5 < r \end{cases} \quad (6)$$

4.6.2. Scott–Knott Effect Size Difference (ESD)

To find the best performing ensemble models, we used the Scott–Knott ESD test proposed by Tantithamthavorn et al. [30,54] to rank the ensemble models. The Scott–Knott algorithm [55] uses hierarchical clustering to cluster the treatments (i.e., the stacked and tree-based ensembles) into distinct groups, whereas the mean differences (i.e., the mean of the AUC scores) between these groups are statistically significant. The algorithm starts with one group containing all treatments. Then, iteratively, it divides the treatments into two distinct groups or merges two groups into one group based on the treatment means. The process is repeated until (1) the mean differences for all treatments within a group are negligible; and (2) the mean differences of treatments between the groups are non-negligible.

5. Results and Discussion

In this section, we present the impacts of hyperparameter optimization on the prediction performances of tree-based ensembles. Next, we compare the prediction performance of the stacking ensemble against those of tree-based ensembles.

5.1. Effects of Hyperparameter Optimization on Tree-Based Ensembles

In order to find the magnitude of change in the prediction performance, we used the Wilcoxon size effect estimate. Table 5 shows the categories of the impact of hyperparameter optimization on the studied tree-based ensembles using the numbers of ensembles thus affected. RF ensembles were highly influenced by hyperparameter optimization, as it tended to have medium to large impact on 50% of the defect datasets. On the contrary, HGB was slightly affected by hyperparameter optimization; it had negligible impact on more than 50% of the ensembles. On average, the magnitude of change was considered negligible to small in the other tree-based ensembles.

Table 5. Wilcoxon effect size for each of the studied tree-based ensembles.

Classifier	Negligible	Small	Medium	Large
Ada	7	7	5	2
CAT	2	11	6	2
ET	8	4	8	1
GB	7	6	7	1
HGB	13	3	2	3
RF	9	1	7	4
XGB	7	9	2	3

The effect size estimates the magnitude of the change; however, it does not determine whether the change is positive or negative. Therefore, we further expanded previous results to show whether hyperparameter optimization increased or decreased the prediction performance. Figure 2 shows the effect size for each tree-based ensemble along with the positive (i.e., in blue) or negative (i.e., in red) impact. Negligible = 0, small = 1, medium = 2 and large = 3. For each ensemble, the total number of positive and negative represent the number of medium and large effect size. Overall, none of the tree-based ensembles showed a positive impact on any datasets. We observed that ET and RF had a negative impact in most of the datasets. Additionally, as shown in the figure, we note that hyperparameter optimization had the lowest impact on HGB, as it had negligible impact across 13 datasets. Clearly, the impact is related to the dataset's characteristics; for example, the prediction

performances on Debug, Eclipse 2.1, Eclipse PDE UI, Lucene 2.4 and Prop 1 datasets deteriorated for all tree-based ensembles after applying hyperparameter optimization. Similarly, on other datasets, including Eclipse 2.0, JM1, Prop 3 and Xalan 2.5, applying hyperparameter optimization increased the prediction performances for all ensembles.

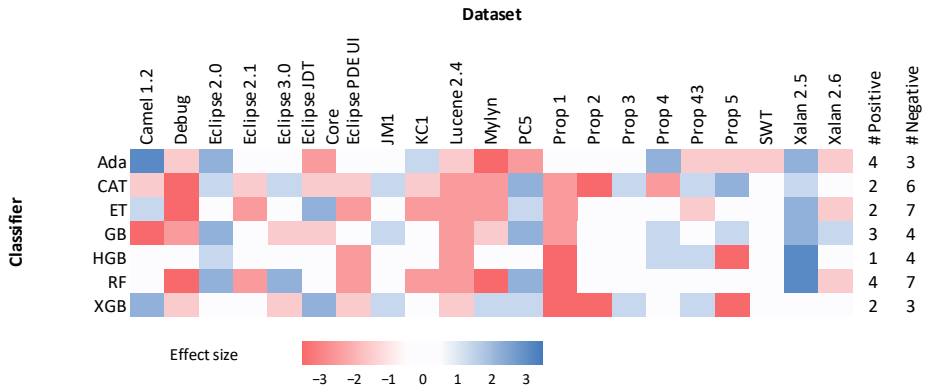

Figure 2. Wilcoxon effect size for each of the studied tree-based ensembles on every dataset.

Computational cost is another important aspect to be considered when performing hyperparameter optimization. For each tree-based ensemble, we computed the amount of time required to optimize the model for each of the studied datasets. Each hyperparameter combination in the grid space was evaluated by 10 independent runs (i.e., 5 × 2-fold cross-validation) to select the tuned model with the best cross-validation score.The computational cost of applying grid search hyperparameter optimization is presented in Table 6. Clearly, CatBoost required a lot of computational effort compared to the other tree-based ensembles. The bagging tree-based ensembles (i.e., RF and ET) took the least amount of time for hyperparameter optimization. Focusing on boosting ensembles, we observe that long computational time was related to the size of the datasets in most cases. However, for the bagging ensembles, we notice that larger datasets, such as JM1, Prop1 and Prop 5, took the least amount of time.

Table 6. The computational cost of applying grid search hyperparameter optimization in milliseconds.

Classifier	ET	RF	Ada	GB	HGB	XGB	CAT
Camel 1.2	13	16	21	138	168	77	1842
Debug	9	12	4	71	78	31	673
Eclipse 2.0	5	7	13	234	366	62	3915
Eclipse 2.1	6	8	15	291	467	80	4358
Eclipse 3.0	5	8	15	342	446	72	5136
Eclipse JDT Core	5	6	10	48	67	30	797
Eclipse PDE UI	5	6	9	59	87	32	918
JM1	20	27	40	930	634	116	7318
KC1	14	23	32	177	260	84	2907
Lucene 2.4	10	14	17	77	82	37	743
Mylyn	11	12	16	135	181	46	1143
PC5	20	26	35	362	406	112	4976
Prop 1	6	8	13	414	321	883	4095
Prop 2	6	8	20	283	287	816	2126
Prop 3	5	7	13	204	251	750	2038
Prop 4	5	7	13	216	263	796	1910
Prop 43	5	7	12	196	231	574	2023

Table 6. *Cont.*

Classifier	ET	RF	Ada	GB	HGB	XGB	CAT
Prop 5	6	9	16	409	308	896	4199
SWT	10	13	17	116	161	45	1321
Xalan 2.5	10	14	16	106	131	45	1239
Xalan 2.6	10	12	16	110	140	42	1313
AVG time (ms)	8.86	11.90	17.29	234.19	254.05	267.90	2618.57

RQ1 Answer. RF ensembles were most impacted when performing hyperparameter optimization. Hyperparameter optimization had the lowest impact on HGB ensembles. The impact of hyperparameter optimization on RF ensemble was negative on most of the datasets. On average, the magnitude of change was considered negligible to small for the other tree-based ensembles. Hyperparameter optimization of CAT was most costly in terms of computational time compared to the other tree-based ensembles.

5.2. Stacking Ensembles

The stacking ensemble prediction performance is compared with those of the tree-based ensembles in terms of F1-score, AUC and Brier in Figure 3. Considering the three evaluation measures, we found that the stacking ensemble's prediction performance was high and consistent across almost all of the datasets. Moreover, the XGB tree-based ensemble showed a competitive prediction performance by having AUC scores equal to or greater than the stacking ensemble in almost 50% of the datasets. On the other hand, Ada was consistently one of worst performing ensembles on all datasets. Two other tree-based ensembles, ET and HGB, were the second worst ensembles; both achieved low AUC and F1 scores. The most significantly different performance was observed for the RF ensembles: low F1 scores on almost all datasets while having high AUC and Brier scores for 50% and 66% of the datasets, correspondingly.

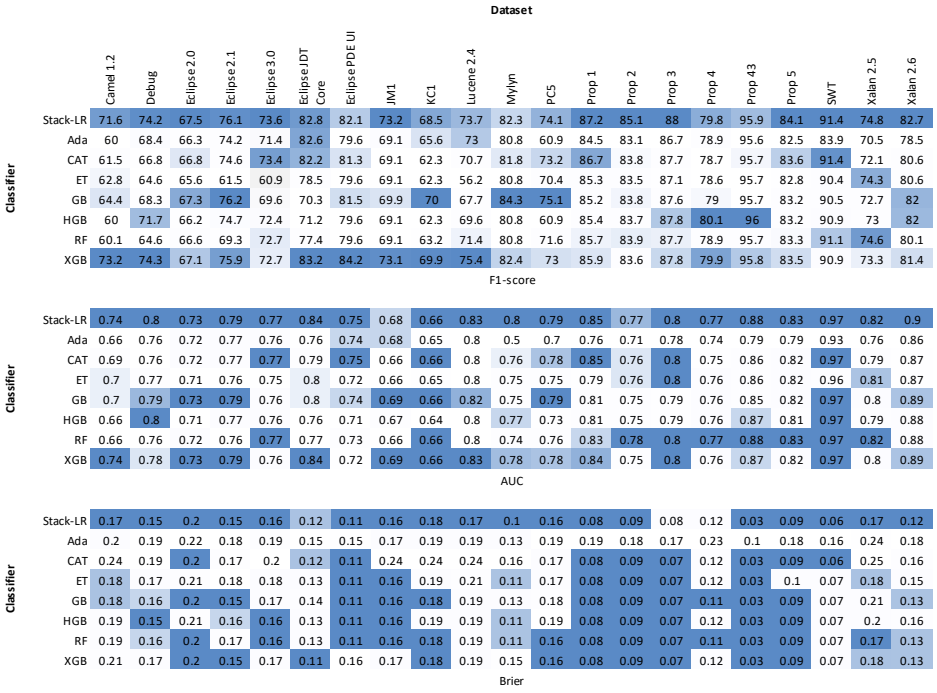

Figure 3. The stacking ensemble versus fine-tuned tree-based ensembles.

The Scott–Knott clusters with the AUC values in each dataset are shown in Figure 4. From the figure, we observe that the stacking ensemble was ranked as the best performing for 80% of the datasets, and the second best for the remaining datasets. Moreover, RF and

XGB were among the first clusters for 40% and 30% of the datasets, respectively. GB and CAT ensembles were among the best performing ensemble models for 23% of the datasets.

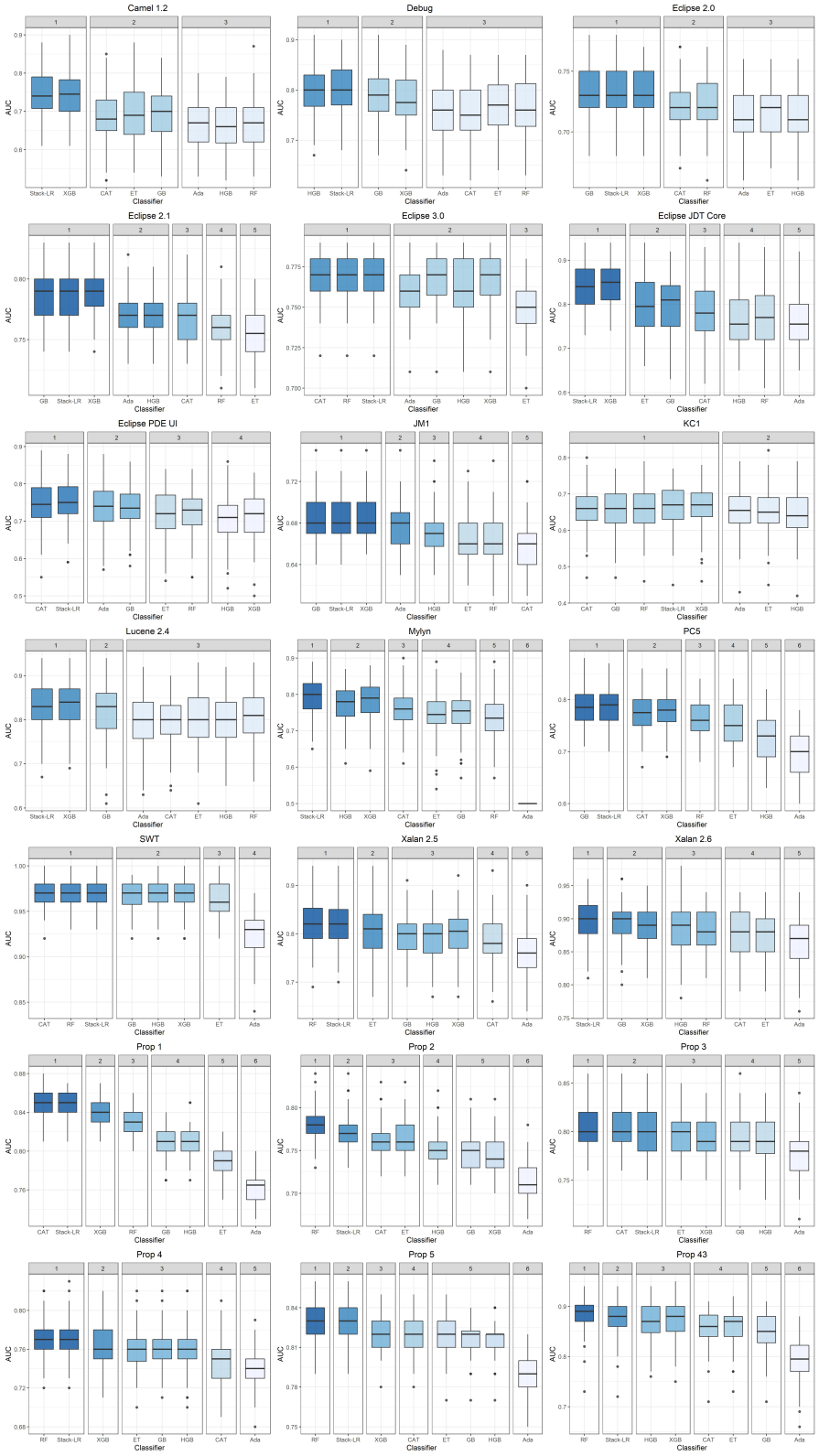

Figure 4. AUC rankings of the studied ensembles models.

We further analyzed the rankings of ensemble models over the 21 datasets by performing a double Scott–Knott test. The procedure is shown in Figure 5. First, for each dataset, we

ranked the ensemble models based on the AUC values using Scott–Knott algorithm. Then, we constructed a new dataset where each ensemble model had 21 ranking values (one from each dataset) obtained from the previous step. The newly constructed data were fed into the Scott–Knott algorithm to rank the ensemble models for all datasets. Figure 6 presents the ranking of the studied ensemble models over the 21 datasets. Stacking ensemble stayed the first-ranked ensemble. Next came RF and XGB as the second-best-performing ensembles.

RQ2 Answer. The stacking generalization ensemble's prediction performance was significantly better, or at least competitive, compared to tree-based ensembles in defect prediction. Among all tree-based ensembles, RF and XGB ensembles showed significantly superior prediction performance.

Figure 5. Double Scott–Knott for ranking ensemble models across multiple datasets.

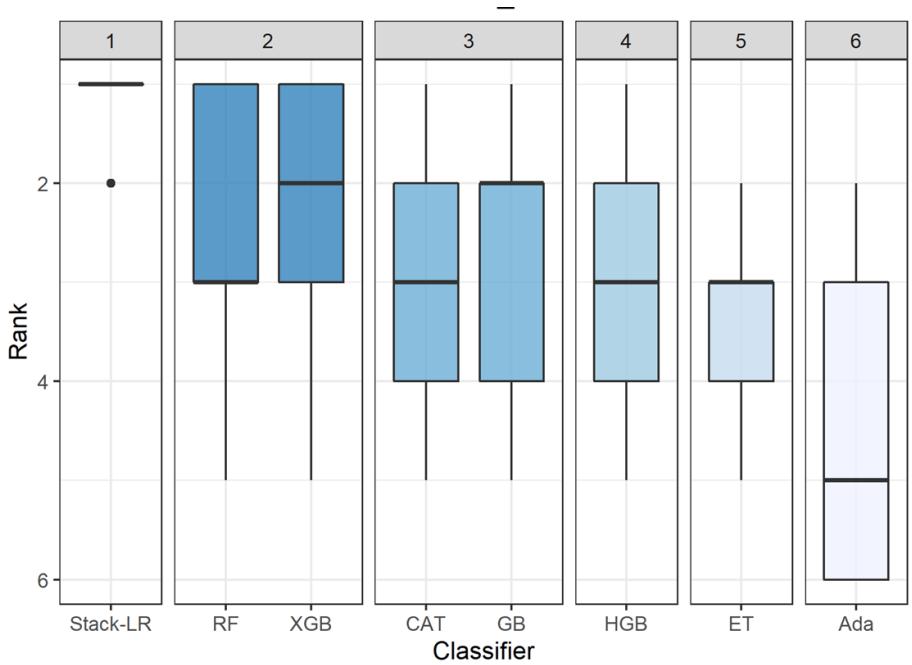

Figure 6. Ranking of the studied ensemble models across all datasets.

5.3. Threats to Validity

We identified and assessed a set of threats to validity to assure the quality of this empirical investigation. Each threat is explained, along with the methods taken to minimize the threats.

5.3.1. Internal Validity

An internal threat was identified in the metrics used as independent variables to predict software defects. We used multiple datasets from several corpora, and each dataset had a different set of metrics and granularity (i.e., method, class or file). Nevertheless, to mitigate this threat, we used several datasets that had the same set of metrics (e.g., SWT and Debug). Moreover, we believe that the diversity of the metrics help strengthen the generalization of the conclusions of this empirical investigation.

5.3.2. Construct Validity

Threats to construct validity concern the relation between the theory and the observations. One threat to construct validity in this experiment was the evaluation metrics used to evaluate the prediction performance. The use of one evaluation measurement raises the risk of bias and contributes to misleading outcomes. To mitigate this threat, we employed three evaluation measures: one threshold-dependent measure (F-measure) and two threshold-independent measures (AUC and Brier) so they could be checked against each other. Moreover, we selected the commonly used measures found in the literature in software defect prediction, so they could be compared with other studies. The selection of the seven tree-based ensembles was the another threat to construct validity that might have led to bias. We mitigate this risk by combining well-known classification techniques (e.g., AdaBoost and random forests) with new techniques that have yet to be thoroughly examined in this area (e.g., extra trees and CatBoost). The final set of classifiers selected in our study included a wide variety of homogeneous ensembles from several categories (i.e., bagging and boosting).

5.3.3. External Validity

External threats relate to the generalization of our empirical study conclusions. Our results are based on the datasets created by several research groups, and the variations in the measurements used to collect the datasets could affect the validity of our results. To mitigate this threat, we selected datasets that cover different application domains and have different sizes and granularity. Additionally, we selected high quality datasets that have high EPV values, which has been shown to produce more stable results [42]. Nevertheless, the empirical study's findings cannot be generalized to other industrial fields. The development of the ensemble models was another threat. In our empirical study we used the scikit-learn library for data preprocessing and building the ensemble models, and coding errors might have occurred. We employed pair programming, an agile software development approach, to assure code quality, and the Google Colab environment for code exchange and execution to mitigate this threat. Moreover, we assessed the code correctness using benchmark datasets from the scikit-learn package (e.g., iris dataset).

5.3.4. Conclusion Validity

Conclusion validity threats are concerned with the relationship between the treatments and the outcomes. We used a variety of statistical comparison tests for comparing ensemble models in this empirical study to examine the effects (i.e., magnitude) of hyperparameter optimization on tree-based ensembles. The underlying assumptions influence the statistical test that should be applied. Any deviation from the statistical test assumptions might result in incorrect results. We utilized the Shapiro–Wilk test for confirming the normality of our data. The non-parametric Wilcoxon effect size test was used because the data were not normally distributed.

6. Conclusions

In this empirical study, we explored the application of a stacking ensemble of fine-tuned tree-based ensembles for software defect prediction. This empirical investigation has two main contributions: (1) we applied hyperparameter optimization on seven tree-based ensembles to explore the impacts of hyperparameter optimization on ensemble models; (2) we built a stacking ensemble of fine-tuned tree-based ensembles and evaluated to what extent it would offer an increase in the prediction performance over fine-tuned tree-based ensembles. The results of this study show the large impact of hyperparameter optimization on extra trees and random forest ensembles. Moreover, the stacking ensemble of fine-tuned tree-based ensembles showed high prediction performance compared to all fine-tuned tree-based ensemble models.

As future work, the use of different heterogeneous ensembles, such as voting, could be investigated and compared with the use of the proposed ensemble models. Furthermore,

the impact of hyperparameter optimization on the stacking ensemble can be explored and different meta-classifiers could be used and compared. Finally, we used grid search to optimize tree-based ensembles hyperparameters and reported our analysis on the computational cost. Future research might be directed to further evaluating the other three hyperparameter optimization techniques (i.e., random search, genetic algorithm and differential evolution) in terms of performance enhancement and computational cost when used to optimize tree-based ensembles hyperparameters.

Author Contributions: A.A.: Conceptualization, methodology, software, data curation, design, analysis, writing of the manuscript. H.A.: conceptualization, methodology, software, data curation, design, analysis, writing of the manuscript. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Utilized software defect datasets are publicly available.

Acknowledgments: The authors would like to acknowledge the support of King Fahd University of Petroleum and Minerals (KFUPM), Saudi Arabia in the development of this work.

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

References

1. Menzies, T.; Milton, Z.; Turhan, B.; Cukic, B.; Jiang, Y.; Bener, A. Defect prediction from static code features: Current results, limitations, new approaches. *Autom. Softw. Eng.* **2010**, *17*, 375–407. https://doi.org/10.1007/s10515-010-0069-5.
2. Malhotra, R. A systematic review of machine learning techniques for software fault prediction. *Appl. Soft Comput.* **2015**, *27*, 504–518. https://doi.org/10.1016/j.asoc.2014.11.023.
3. Aljamaan, H.; Alazba, A. Software defect prediction using tree-based ensembles. In Proceedings of the 16th ACM International Conference on Predictive Models and Data Analytics in Software Engineering, Virtual Event, 8–9 November 2020; Association for Computing Machinery: New York, NY, USA, 2020; pp. 1–10.
4. Tosun, A.; Bener, A. Reducing false alarms in software defect prediction by decision threshold optimization. In Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement, Lake Buena Vista, FL, USA, 15 October 2009; pp. 477–480, ISSN 1949–3789. https://doi.org/10.1109/ESEM.2009.5316006.
5. Tantithamthavorn, C.; McIntosh, S.; Hassan, A.E.; Matsumoto, K. Automated Parameter Optimization of Classification Techniques for Defect Prediction Models. In Proceedings of the 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), Austin, TX, USA, 14–22 May 2016; pp. 321–332, ISSN 1558–1225. https://doi.org/10.1145/2884781.2884857.
6. Osman, H.; Ghafari, M.; Nierstrasz, O. Hyperparameter optimization to improve bug prediction accuracy. In Proceedings of the 2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE), Klagenfurt, Austria, 21 February 2017; pp. 33–38. https://doi.org/10.1109/MALTESQUE.2017.7882014.
7. Rokach, L. Ensemble-based classifiers. *Artif. Intell. Rev.* **2010**, *33*, 1–39. https://doi.org/10.1007/s10462-009-9124-7.
8. Zhou, Z.H. *Ensemble Methods: Foundations and Algorithms*, 1st ed.; Chapman & Hall/CRC: Boca Raton, FL, USA, 2012.
9. Safavian, S.; Landgrebe, D. A survey of decision tree classifier methodology. *IEEE Trans. Syst. Man Cybern.* **1991**, *21*, 660–674. https://doi.org/10.1109/21.97458.
10. Freund, Y. Boosting a Weak Learning Algorithm by Majority. *Inf. Comput.* **1995**, *121*, 256–285.
11. Breiman, L. Bagging predictors. *Mach. Learn.* **1996**, *24*, 123–140.
12. Ho, T.K. Random decision forests. In Proceedings of the 3rd International Conference on Document Analysis and Recognition, Montreal, QC, Canada, 14–16 August 1995; Volume 1, pp. 278–282. https://doi.org/10.1109/ICDAR.1995.598994.
13. Geurts, P.; Ernst, D.; Wehenkel, L. Extremely randomized trees. *Mach. Learn.* **2006**, *63*, 3–42. https://doi.org/10.1007/s10994-006-6226-1.
14. Friedman, J.H. Greedy Function Approximation: A Gradient Boosting Machine. *Ann. Stat.* **2001**, *29*, 1189–1232.
15. Guryanov, A. Histogram-Based Algorithm for Building Gradient Boosting Ensembles of Piecewise Linear Decision Trees. In *Analysis of Images, Social Networks and Texts*; Lecture Notes in Computer Science; van der Aalst, W.M.P., Batagelj, V., Ignatov, D.I., Khachay, M., Kuskova, V., Kutuzov, A., Kuznetsov, S.O., Lomazova, I.A., Loukachevitch, N., Napoli, A., et al., Eds.; Springer International Publishing: Cham, Switzerland, 2019; pp. 39–50. https://doi.org/10.1007/978-3-030-37334-4_4.
16. Chen, T.; Guestrin, C. XGBoost: A Scalable Tree Boosting System. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16), San Francisco, CA, USA, 13–17 August 2016; Association for Computing Machinery: San Francisco, CA, USA, 2016; pp. 785–794. https://doi.org/10.1145/2939672.2939785.

17. Dorogush, A.V.; Ershov, V.; Gulin, A. CatBoost: Gradient boosting with categorical features support. *arXiv* **2018**, arXiv:1810.11363.
18. Wolpert, D.H. Stacked generalization. *Neural Netw.* **1992**, *5*, 241–259. https://doi.org/10.1016/S0893-6080(05)80023-1.
19. Aggarwal, C.C. *Data Classification: Algorithms and Applications*; Google-Books-ID: NwQZCwAAQBAJ; CRC Press: Boca Raton, FL, USA, 2015.
20. Aljamaan, H.I.; Elish, M.O. An empirical study of bagging and boosting ensembles for identifying faulty classes in object-oriented software. In Proceedings of the 2009 IEEE Symposium on Computational Intelligence and Data Mining, Nashville, TN, USA, 30 March–2 April 2009; pp. 187–194. https://doi.org/10.1109/CIDM.2009.4938648.
21. Yohannese, C.W.; Li, T.; Simfukwe, M.; Khurshid, F. Ensembles based combined learning for improved software fault prediction: A comparative study. In Proceedings of the 2017 12th International Conference on Intelligent Systems and Knowledge Engineering (ISKE), NanJing, China, 24–26 November 2017; pp. 1–6. https://doi.org/10.1109/ISKE.2017.8258836.
22. Petrić, J.; Bowes, D.; Hall, T.; Christianson, B.; Baddoo, N. Building an Ensemble for Software Defect Prediction Based on Diversity Selection. In Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '16), Ciudad Real, Spain, 8–9 September 2016; Association for Computing Machinery: Ciudad Real, Spain, 2016; pp. 1–10. https://doi.org/10.1145/2961111.2962610.
23. Pandey, S.K.; Mishra, R.B.; Tripathi, A.K. BPDET: An effective software bug prediction model using deep representation and ensemble learning techniques. *Expert Syst. Appl.* **2020**, *144*, 113085. https://doi.org/10.1016/j.eswa.2019.113085.
24. Hussain, S.; Keung, J.; Khan, A.A.; Bennin, K.E. Performance Evaluation of Ensemble Methods For Software Fault Prediction: An Experiment. In Proceedings of the ASWEC 2015 24th Australasian Software Engineering Conference (ASWEC '15), Adelaide, SA, Australia, 28 September–1 October 2015; Association for Computing Machinery: Adelaide, SA, Australia, 2015; Volume II, pp. 91–95. https://doi.org/10.1145/2811681.2811699.
25. Li, R.; Zhou, L.; Zhang, S.; Liu, H.; Huang, X.; Sun, Z. Software Defect Prediction Based on Ensemble Learning. In Proceedings of the 2019 2nd International Conference on Data Science and Information Technology (DSIT 2019), Seoul, Korea, 19–21 July 2019; Association for Computing Machinery: Seoul, Korea, 2019; pp. 1–6. https://doi.org/10.1145/3352411.3352412.
26. Tran, H.D.; Hanh, L.T.M.; Binh, N.T. Combining feature selection, feature learning and ensemble learning for software fault prediction. In Proceedings of the 2019 11th International Conference on Knowledge and Systems Engineering (KSE), Da Nang, Vietnam, 24–26 October 2019; pp. 1–8. ISSN 2164–2508, https://doi.org/10.1109/KSE.2019.8919292.
27. Tong, H.; Liu, B.; Wang, S. Software defect prediction using stacked denoising autoencoders and two-stage ensemble learning. *Inf. Softw. Technol.* **2018**, *96*, 94–111. https://doi.org/10.1016/j.infsof.2017.11.008.
28. Laradji, I.H.; Alshayeb, M.; Ghouti, L. Software defect prediction using ensemble learning on selected features. *Inf. Softw. Technol.* **2015**, *58*, 388–402. https://doi.org/10.1016/j.infsof.2014.07.005.
29. Khan, F.; Kanwal, S.; Alamri, S.; Mumtaz, B. Hyper-Parameter Optimization of Classifiers, Using an Artificial Immune Network and Its Application to Software Bug Prediction. *IEEE Access* **2020**, *8*, 20954–20964. https://doi.org/10.1109/ACCESS.2020.2968362.
30. Tantithamthavorn, C.; McIntosh, S.; Hassan, A.E.; Matsumoto, K. The Impact of Automated Parameter Optimization on Defect Prediction Models. *IEEE Trans. Softw. Eng.* **2019**, *45*, 683–711. https://doi.org/10.1109/TSE.2018.2794977.
31. Fu, W.; Menzies, T.; Shen, X. Tuning for Software Analytics: Is it Really Necessary? *Inf. Softw. Technol.* **2016**, *76*, 135–146. https://doi.org/10.1016/j.infsof.2016.04.017.
32. Öztürk, M.M. Comparing Hyperparameter Optimization in Cross- and Within-Project Defect Prediction: A Case Study. *Arab. J. Sci. Eng.* **2019**, *44*, 3515–3530. https://doi.org/10.1007/s13369-018-3564-9.
33. Basili, V.; Rombach, H. The TAME project: Towards improvement-oriented software environments. *IEEE Trans. Softw. Eng.* **1988**, *14*, 758–773. https://doi.org/10.1109/32.6156.
34. 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. https://doi.org/10.1109/TSE.2013.11.
35. D'Ambros, M.; Lanza, M.; Robbes, R. An extensive comparison of bug prediction approaches. In Proceedings of the 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010), Cape Town, South Africa, 2–3 May 2010; pp. 31–41. ISSN 2160–1860. https://doi.org/10.1109/MSR.2010.5463279.
36. Jureczko, M.; Madeyski, L. Towards identifying software project clusters with regard to defect prediction. In Proceedings of the 6th International Conference on Predictive Models in Software Engineering (PROMISE '10), Timisoara, Romania, 12–13 September 2010; Association for Computing Machinery: New York, NY, USA, 2010; pp. 1–10. https://doi.org/10.1145/1868328.1868342.
37. Wu, R.; Zhang, H.; Kim, S.; Cheung, S.C. ReLink: Recovering links between bugs and changes. In Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11), Szeged, Hungary, 5–9 September 2011; Association for Computing Machinery: New York, NY, USA, 2011; pp. 15–25. https://doi.org/10.1145/2025113.2025120.
38. Zimmermann, T.; Premraj, R.; Zeller, A. Predicting Defects for Eclipse. In Proceedings of the Third International Workshop on Predictor Models in Software Engineering (PROMISE '07), Minneapolis, MN, USA, 20 May 2007; IEEE Computer Society: Minneapolis, MN, USA, 2007; p. 9. https://doi.org/10.1109/PROMISE.2007.10.
39. Kim, S.; Zhang, H.; Wu, R.; Gong, L. Dealing with noise in defect prediction. In Proceedings of the 2011 33rd International Conference on Software Engineering (ICSE), Honolulu, HI, USA, 21–28 May 2011; pp. 481–490, ISSN 1558–1225. https://doi.org/10.1145/1985793.1985859.

40. Turhan, B.; Menzies, T.; Bener, A.B.; Di Stefano, J. On the relative value of cross-company and within-company data for defect prediction. *Empir. Softw. Eng.* **2009**, *14*, 540–578. https://doi.org/10.1007/s10664-008-9103-7.
41. Peduzzi, P.; Concato, J.; Kemper, E.; Holford, T.R.; Feinstein, A.R. A simulation study of the number of events per variable in logistic regression analysis. *J. Clin. Epidemiol.* **1996**, *49*, 1373–1379. https://doi.org/10.1016/s0895-4356(96)00236-3.
42. Tantithamthavorn, C.; McIntosh, S.; Hassan, A.E.; Matsumoto, K. An Empirical Comparison of Model Validation Techniques for Defect Prediction Models. *IEEE Trans. Softw. Eng.* **2017**, *43*, 1–18. https://doi.org/10.1109/TSE.2016.2584050.
43. Bergstra, J.; Bardenet, R.; Bengio, Y.; Kégl, B. Algorithms for hyper-parameter optimization. In *Proceedings of the 24th International Conference on Neural Information Processing Systems*; Curran Associates Inc.: Red Hook, NY, USA, 12–15 December 2011; pp. 2546–2554.
44. Dietterich, T.G. Approximate statistical tests for comparing supervised classification learning algorithms. *Neural Comput.* **1998**, *10*, 1895–1923.
45. Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; et al. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* **2011**, *12*, 2825–2830.
46. Jiang, Y.; Cukic, B.; Menzies, T. Can data transformation help in the detection of fault-prone modules? In *Proceedings of the 2008 Workshop on Defects in Large Software Systems (DEFECTS '08)*; Association for Computing Machinery: New York, NY, USA, 20 July 2008; pp. 16–20. https://doi.org/10.1145/1390817.1390822.
47. Young, W.; Weckman, G.; Holland, W. A survey of methodologies for the treatment of missing values within datasets: Limitations and benefits. *Theor. Issues Ergon. Sci.* **2011**, *12*, 15–43. https://doi.org/10.1080/14639220903470205.
48. Mundfrom, D.J.; Whitcomb, A. *Imputing Missing Values: The Effect on the Accuracy of Classification*; Multiple Linear Regression Viewpoints: Birmingham, UK, 1998; Volume 25.
49. Kohavi, R. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence-Volume 2 (IJCAI'95)*; Morgan Kaufmann Publishers Inc.: Montreal, QC, Canada, 20–25 August 1995; pp. 1137–1143.
50. Qu, Y.; Zheng, Q.; Chi, J.; Jin, Y.; He, A.; Cui, D.; Zhang, H.; Liu, T. Using K-core Decomposition on Class Dependency Networks to Improve Bug Prediction Model's Practical Performance. *IEEE Trans. Softw. Eng.* **2019**, *47*, 348–366.
51. Brier, G.W. Verification of Forecasts Expressed in Terms of Probability. *Mon. Weather Rev.* **1950**, *78*, 1–3. https://doi.org/10.1175/1520-0493(1950)078<0001:VOFEIT>2.0.CO;2.
52. Fritz, C.O.; Morris, P.E.; Richler, J.J. Effect size estimates: Current use, calculations, and interpretation. *J. Exp. Psychol. Gen.* **2012**, *141*, 2–18. https://doi.org/10.1037/a0024338.
53. Cohen, J. A power primer. *Psychol. Bull.* **1992**, *112*, 155–159. https://doi.org/10.1037//0033-2909.112.1.155.
54. Tantithamthavorn, C. ScottKnottESD: The Scott-Knott Effect Size Difference (ESD) Test. 2018. Available online: https://cran.r-project.org/package=ScottKnottESD (accessed on 2 March 2022).
55. Scott, A.J.; Knott, M. A Cluster Analysis Method for Grouping Means in the Analysis of Variance. *Biometrics* **1974**, *30*, 507–512. https://doi.org/10.2307/2529204.