IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 31 January 2024, accepted 17 March 2024, date of publication 21 March 2024, date of current version 16 April 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3380024

RESEARCH ARTICLE

Hyperparameter Optimization for Software Bug Prediction Using Ensemble Learning

DIMAH AL-FRAIHAT[1], YOUSEF SHARRAB[2], ABDEL-RAHMAN AL-GHUWAIRI[3], HAMZEH ALSHISHANI[3], AND ABDULMOHSEN ALGARNI[4]

[1]Department of Software Engineering, Faculty of Information Technology, Isra University, Amman 11622, Jordan
[2]Department of Data Science and Artificial Intelligence, Faculty of Information Technology, Isra University, Amman 11622, Jordan
[3]Department of Software Engineering, Faculty of Prince Al-Hussien Bin Abdullah for IT, Hashemite University, Zarqa 13133, Jordan
[4]Department of Computer Science, King Khalid University, Abha 61421, Saudi Arabia

Corresponding author: Dimah Al-Fraihat (d.fraihat@iu.edu.jo)

This work was supported by the Deanship of Scientific Research, King Khalid University, under Grant R.G.P.2/93/45.

ABSTRACT Software Bug Prediction (SBP) is an integral process to the software's success that involves predicting software bugs before their occurrence. Detecting software bugs early in the development process enhances software quality, performance, and reduces software costs. The integration of Machine Learning (ML) algorithms has significantly improved software bug prediction accuracy and concurrently reduced costs and resource utilization. Numerous studies have explored the impact of Hyperparameter Optimization on single classifiers, enhancing these models' overall performance in SBP analysis. Ensemble Learning (EL) approaches have also demonstrated increased model accuracy and performance on SBP datasets. This study proposes a novel learning model for predicting software bugs through the utilization of EL and tuning hyperparameters. The results are compared with single hypothesis learning models using the WEKA software. The dataset, collected by the National Aeronautics and Space Administration (NASA) U.S.A., comprises 10,885 instances with 20 attributes, including a classifier for defects in one of their coding projects. The findings indicate that EL models outperform single hypothesis learning models, and the proposed model's accuracy increases after optimization. Furthermore, the accuracy of the proposed model demonstrates improvement following the optimization process. These results underscore the efficacy of ensemble learning, coupled with hyperparameter optimization, as a viable approach for enhancing the predictive capabilities of software bug prediction models.

INDEX TERMS Software bug prediction, machine learning, hyperparameter optimization, ensemble learning.

I. INTRODUCTION

Software bugs increase the cost of the Software Development Life Cycle (SDLC), as the expense of software development rises proportionately with the time of discovery [1]. This impact is particularly significant if bugs are identified after the software release, beginning to affect the end-user experience. Therefore, it is always best to find bugs as soon as they occur. Bugs also influence software quality and reliability, prompting many software companies to research methods for identifying bugs as soon as they occur. This research consumes resources and contributes to delays in release dates. All of these considerations have led to the development of Software Bug Prediction (SBP) systems, which assist software development organizations in automating the identification of bugs before they occur, thereby preventing the wastage of resources.

Predicting the emergence of bugs and understanding the Software Development Life Cycle (SDLC) process flow are critical factors that contribute to performance criteria [2]. The application of ML algorithms for defect prediction is versatile, as it can be employed across various stages of the SDLC, including problem detection, planning, design, building, testing, deployment, and maintenance. This application extends

The associate editor coordinating the review of this manuscript and approving it for publication was Ahmed M. Elmisery.

to multiple SDLC models, such as the Waterfall Model, Iterative Model, Agile Model, V-Shaped Model, Spiral Model, and Big Bang Model. ML algorithms prove effective in determining the presence of bugs in a given piece of code. Enhanced programming methods can significantly elevate the overall quality of the program. ML represents one of the most rapidly expanding fields of computer science, with far-reaching applications. It involves the automatic identification of meaningful patterns in records. The primary goal of ML tools is to provide algorithms with the opportunity to learn and adapt [3], [4].

With the rise of ML Algorithms in the past couple of years, many companies utilized ML to find and predict software bugs before occurring. ML Algorithms help organizations make future predictions in many fields using mathematical equations with statistics and ML algorithms. There are two types of ML algorithms, supervised learning and unsupervised learning. In supervised learning, the algorithm's input and output are based on historical data related to the study, which is used to train the algorithms and find patterns; these patterns are then used to predict future outputs. In unsupervised learning, the learning algorithms process data to find patterns in unlabeled fields. Early bug discovery and prediction are critical exercises in the software development life cycle [5]. They contribute to a higher satisfaction rate among users and enhance the overall performance and quality of the software project.

Defect prediction models can guide testing efforts towards code that is susceptible to bugs. Before software delivery to the customer, latent bugs in the code may surface, and once identified, these flaws can be rectified prior to delivery at a fraction of the cost compared to post-delivery repairs [6]. Code defects incur significant expenses for companies each year, with billions of dollars spent on identification and correction. Models that accurately pinpoint the locations of bugs in code have the potential to save businesses substantial amounts of money. Given the high costs involved, even minor improvements in detecting and repairing flaws can have a significant impact on total costs.

Many studies have been conducted on Bug Detection using ML Algorithms. Some of these studies have utilized EL, which enables researchers and practitioners to employ more than one machine learning algorithm. By employing different ML algorithms with diverse methodologies, EL allows for the aggregation of votes from multiple ML algorithms used together, enhancing the ability to make more accurate predictions. Further, previous research studies have examined different types of algorithms to predict software bugs. Although some of these proposed algorithms showed higher accuracy rates, most of the cited bug prediction studies for software projects had their settings values set to default values [7].

Hyperparameter optimization can enhance accuracy and improve the performance of ML models. The optimization of hyperparameters in Bayesian, SVM, and KNN models has demonstrated an increase in the accuracy of bug prediction models, as observed by Osman et al. [8] and Levesque et al. [9]. Hyperparameter optimizers are utilized to fine-tune the control parameters of data mining algorithms. It is widely recognized that such tuning enhances classification tasks, including software bug prediction or text classification [10].

To enhance the effectiveness of software bug prediction, this research addresses two key questions. Firstly, it explores whether Ensemble Learning (EL) models outperform single learning models in predicting software bugs. Secondly, the study investigates the impact of adding Hyperparameter Optimization approaches on EL model performance, examining whether this enhancement leads to improved accuracy. These questions serve as the foundation of our exploration, guiding us to understand the nuances of predictive modeling in software bug detection.

The remainder of the paper is organized as follows: Section II presents related work; Section III covers the background of the study, which includes Hyperparameter optimization and ensemble learning; Section IV outlines the methodology, encompassing the dataset, tools used, research questions, and evaluation measures; Section V presents the results and discussion; Section VI provides the conclusion and outlines future work.

II. RELATED WORK

This section navigates through recent contributions in the dynamic field of software bug prediction by exploring various methodologies and approaches used by previous researchers. These studies enrich our understanding of effective bug prediction models and offer diverse insights and methodologies to address the challenges posed by software defects.

In the study of Hammouri et al. [5], the researchers conducted a thorough examination of three supervised ML models—Naive Bayes, Decision Tree, and Artificial Neural Networks—specifically tailored for predicting software bugs. Their research involved a meticulous comparison with two previously proposed models, revealing that the three models exhibited a significantly higher accuracy rate than the proposed two models.

Di Nucci et al. [11] undertook an innovative approach, quantifying the dispersal of modifications made by developers working on a component, and utilized this data to construct a bug prediction model. This study, based on prior research that examined the human factor in bug generation, demonstrates that non-focused developers are more prone to introducing defects compared to their focused counterparts. Notably, the model exhibited superiority when compared to four competitive techniques.

Khan et al. [12] introduced a model for predicting software bugs, integrating ML classifiers with Artificial Immune Networks (AIN) to optimize hyperparameters for enhanced bug prediction accuracy. Employing seven ML algorithms on a bug prediction dataset, the study revealed that hyperparameter tuning of ML classifiers using AIN surpassed the performance of classifiers with default hyperparameters in the context of software bug prediction.

Yang et al. [13] proposed a two-layer Ensemble Learning (TLEL) technique, incorporating both decision trees and Ensemble Learning to enhance just-in-time software defect prediction. The proposed model demonstrated a substantial improvement over existing state-of-the-art ML models. Remarkably, by reviewing only 20% of the code lines, TLEL successfully identified over 70% of the bugs.

Additionally, Ensemble Learning (EL) not only enhances the accuracy of ML models but also improves their overall performance. Balogun et al. [14] scrutinized the performance of individual ML classifiers compared to EL, incorporating the Analytic Network Process (ANP) in their evaluation. The results demonstrated that Ensemble Learning (EL) models achieved a higher priority level than all single classifier methods across 11 software defect datasets.

In a study conducted by Qiu et al. [15], 15 imbalanced EL methods were examined across 31 open-source projects. The findings indicated that imbalanced EL models, which combined both under-sampling and bagging approaches, demonstrated a higher effectiveness rate.

Abdou and Darwish [16] evaluated EL models and introduced the resample technique with Boosting, Bagging, and Rotation Forest. The study concludes that the accuracy improved using EL approaches more than single classifiers. The researchers also noticed that the accuracy and performance improved using Multilayer Perceptron and partial decision trees using Bagging. The study also does not recommend using Support Vector Machine and Logistics as a single classifier with Boosting, Bagging, and Rotation Forest.

Furthermore, Pandey et al. [17] suggested a rudimentary classification-based method for SBP using EL techniques and Deep Representation. The approach encompasses both EL and deep representation in combination. In comparison to EL and other state-of-the-art techniques, the proposed model exhibited superior performance across the majority of the datasets.

In summary, the reviewed literature has provided a diverse range of methodologies for software bug prediction, encompassing supervised machine learning, Ensemble Learning, and deep representation. The comparative analyses underscore the effectiveness of EL models and the importance of hyperparameter optimization in bug prediction. As we transition to the background section, these insights form a solid foundation for exploring the key components of Hyperparameter Optimization and Ensemble Learning in software bug prediction.

III. BACKGROUND

In this section, two crucial components integral to this paper's methodology are examined—Hyperparameter Optimization and Ensemble Learning (EL). This comprehensive exploration aims to provide a detailed understanding of the roles and significance of EL and Hyperparameter Optimization in the context of the research presented in this study.

A. HYPERPARAMETER OPTIMIZATION

In the field of machine learning, the term "optimization" is employed to describe the process of fine-tuning hyperparameters. This involves adjusting parameters such as regularization, kernels, and learning intensity [18]. Hyperparameters play a critical role in enhancing the accuracy of ML classifiers, influencing them throughout the phases of learning, construction, and evaluation. The terms "model discovery" and "hyperparameter tuning" encapsulate the process of identifying the optimal hyperparameters for ML classifiers [12]. As each algorithm comes with a set of parameters, many of which are optimizable, adjusting these parameters becomes pivotal. This optimization aims to achieve a higher model score and enhance the overall performance of the classifier by selecting the optimal configuration for its parameters.

Tuning machine learning models exemplifies an optimization challenge, wherein a diverse set of hyperparameters needs to be fine-tuned to identify the optimal combination of values that minimizes the function's loss or maximizes accuracy. One method employed for hyperparameter optimization is the Manual Search. In this approach, practitioners rely on their expertise to select hyperparameters, followed by model training, accuracy assessment, and iterative refinement. This cyclic process continues until the model's performance and accuracy align with the specified requirements.

Grid search, also referred to as parameter sweep, constitutes an exhaustive approach to hyperparameter optimization, involving a systematic exploration of a predefined subset within a learning algorithm's hyperparameter space. This algorithmic process is guided by an output criterion, often determined through cross-validation on the training set or assessment on a held-out validation set. Cross-validation stands out as one of the most prevalent methods for selecting tuning parameters, identifying the parameter value associated with the lowest cross-validated ranking [19]. Cross-validation parameter selection accommodates the optimization of an infinite number of parameters, allowing researchers to choose a combination of hyperparameter values that the model tests and refines iteratively. Upon completion of the model build, the hyperparameters yielding the highest accuracy score are selected.

B. ENSEMBLE LEARNING

EL algorithms differ from single hypothesis learning algorithms in their unique approach. Rather than relying on a singular, superior explanation for interpreting results, EL algorithms generate a series of hypotheses and use them collectively to determine the labeling of new data points in a diverse manner [20]. Experimental data indicates that ensemble approaches are notably more reliable than individual theories. In a study by Freund et al. [21], efficiency improved in 22 benchmark questions, remained equal in one, and exhibited a decrease in performance in four instances. This highlights the robustness and effectiveness of EL in diverse scenarios.

An ensemble consists of a collection of learners referred to as base learners. The generalization ability of an ensemble typically surpasses that of individual base learners. Ensemble Learning (EL) is particularly appealing because it has the capability to elevate slow learners, which are only marginally better than random guesses, to proficient learners capable of making highly accurate predictions [22]. Consequently, the term ''weak learners'' is often synonymous with ''base learners.'' It is noteworthy, however, that while theoretical studies primarily focus on weak learners, the base learners employed in practical applications are not exclusively weak. Utilizing base learners that are less weak also contributes to superior outcomes.

Among the EL methods, commonly utilized ones include Bagging, Adaboost, Stacking, and Voting. Bagging, short for bootstrap aggregation, stands out as one of the oldest, most fundamental, and perhaps simplest ensemble-based algorithms, demonstrating remarkably high performance [23]. In bagging, diversity of classifiers is achieved by creating bootstrapped replicas of the training data [24]. This involves randomly selecting separate subsets of training data from the entire training dataset, with replacement. Each subset of training data is then employed to train a specific classifier. The classifiers are subsequently aggregated using a majority voting mechanism, where the ensemble decision is determined by the class selected by the majority of classifiers for a given case.

Adaboost operates by training models in successive rounds, with a new model trained in each iteration. At the end of each round, misclassified examples are identified, and their significance is amplified in a new training sequence. This updated training data is then fed back into the beginning of the subsequent round, where a new model is prepared [25]. The underlying theory is that subsequent iterations should compensate for mistakes made by previous models, leading to an overall improvement in the ensemble's performance.

Bagging is particularly effective for unpredictable models that exhibit varying generalization behavior with slight changes in the training data [25]. These models, often termed high variance models, encompass examples like Decision Trees and Neural Networks. However, bagging may not be suitable for extremely simple models. In essence, bagging randomly selects from the set of possible models to create an ensemble of relatively simple models, resulting in predictions that are approximately identical (low diversity).

Voting serves as a technique for consolidating the outputs of multiple classifiers. Three types of voting methods exist: unanimous voting, majority voting, and plurality voting [26]. Unanimous voting takes place when all classifiers unanimously agree on the final judgment. Majority voting occurs when more than half of the votes contribute to the evaluation, while plurality voting happens when the majority of votes determine the final prediction. Each voting method provides a distinct approach to aggregating the decisions of individual classifiers in an ensemble.

Random Forests employs a similar bagging technique to generate random samples of training sets for each Random Tree, known as bootstrap samples. Each current training batch is derived from the original dataset with modifications [27]. Consequently, the tree is constructed using the current set and a randomly generated collection of attributes. The node is divided using the optimal split on the randomly chosen properties. Importantly, established trees in Random Forests are not trimmed, preserving their original structure.

The upcoming methodology section aims to illustrate the practical implementation of Hyperparameter Optimization and Ensemble Learning in the context of software bug prediction.

IV. METHODOLOGY

The classification problem is a common formulation of the supervised learning task in which the learner is expected to learn (approximate the action of) a function that maps a vector into one of several classes by inspecting several input-output examples of the function [4]. Supervised ML is the process of constructing a classification algorithm that can successfully learn from fresh data. Figure 1 demonstrates the steps followed in the application of supervised machine learning to address our optimization problem.

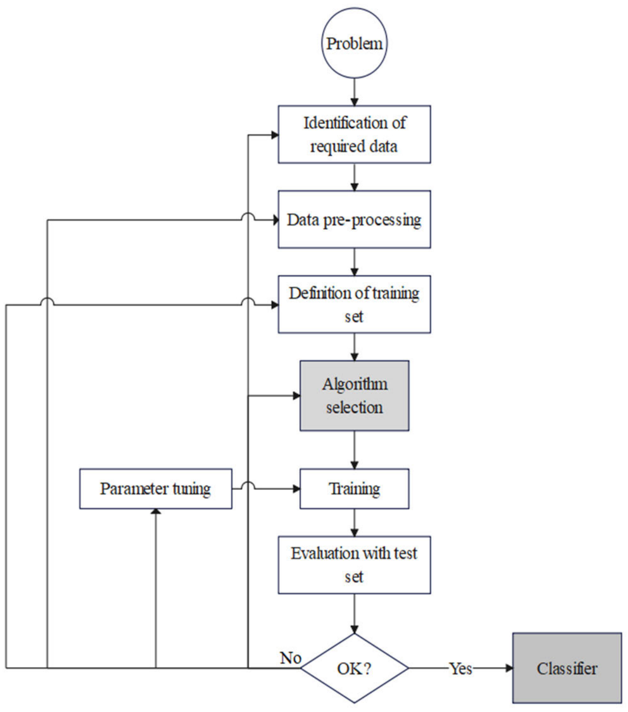

FIGURE 1. Research methodological steps.

The following sub-sections provide a description of the dataset utilized in this study, details the software and tools employed, and presents the evaluation metrics used to assess the performance of the implemented approach.

A. DATASET

In this study, we examined a dataset obtained from the National Aeronautics and Space Administration (NASA), widely recognized as a prevalent resource in the field of software bug prediction. The JM1 dataset was selected from the NASA repository, recognized as one of the largest repositories, with 19% of its data indicating defects. The dataset encompasses 20 attributes and comprises 10,885 instances related to software bugs. Table 1 presents the list of attributes within the dataset, accompanied by concise descriptions of each field and the respective data types stored in each.

TABLE 1. List of dataset attributes.

Attribute	Description	Type
LoC	McCabe's - Line count of code	Numeric
V(g)	McCabe - Cyclomatic complexity	Numeric
ev(g)	McCabe - Essential complexity	Numeric
iv(g)	McCabe - Design complexity	Numeric
N	Halstead total operators and operands	Numeric
V	Halstead – Volume	Numeric
L	Halstead - Program length	Numeric
D	Halstead – Difficulty	Numeric
I	Halstead – Intelligence	Numeric
E	Halstead – Effort	Numeric
T	Halstead's - Time estimator	Numeric
lOCode	Halstead's line count	Numeric
lOComment	Halstead's count of lines of comments	Numeric
lOBlank	Halstead's count of blank lines	Numeric
lOCodeAnd Comment	Halstead's count of blank and code lines	Numeric
uniqOp	Halstead's total number of unique operators	Numeric
uniqOpnd	Halstead's total number of unique operands	Numeric
totalOp	Halstead's total number of operators	Numeric
totalOpnd	Halstead's total number of operands	Numeric
Defects	Module has/has not one or more defects	Boolean

B. HYPERMETER OPTIMIZATION

In this study, the researchers opted for Cross-Validation Parameter Selection (CVParameter Selection), employing a predefined set of hyperparameter values for multiple learning algorithms. CVParameter Selection involves systematically trying out different hyperparameter values for multiple learning algorithms through cross-validation to explore the model's performance under different configurations, ultimately selecting the hyperparameter values that result in the best model performance. Each algorithm has distinct hyperparameters chosen to maximize accuracy and performance on the software bug prediction dataset. The set of hyperparameters applied across various algorithms and their tuned values are shown in Table 2. The hyperparameters included:

- Percentage of weight mass to base training on.
- Use resampling for boosting.
- Random number seed.
- The number of iterations.
- Size of each bag, as a percentage of the training set size.
- Cross-validation folds.

TABLE 2. Hyperparameter optimization results for classifiers.

Classifier	PARAMETER	Default value	Optimized value
Vote	Percentage of weight mass to base training on	1	0.8
Bagging	Size of each bag	1.0	0.8
	Percentage of weight mass to base training on	1.0	0.8
	Use resampling for reboosting	False	True
	Random number seed	42	56
	The number of iterations	50	75
	Cross -validation folds	5	10
Random Forest	Random number seed	42	67
	Percentage of weight mass to base training on	1.0	0.8
	Use resampling for boosting	False	True
	The number of iterations	100	150
	Size of each bag	1	0.7
	Cross -validation folds	5	8
AdaBoostM1	The number of iterations	50	100
	Percentage of weight mass to base training on	1	0.9
	Use resampling for boosting	True	False
	Random number seed	42	89
	Size of each bag	1	0.9
	Cross -validation folds	5	7
Logistics Regression	Use resampling for boosting	False	True
SVM	Cross-validation folds	5	10

C. TOOL

In this study, we employed Weka (Waikato Environment for Knowledge Analysis), a renowned Java-based machine learning software suite developed at the University of Waikato in New Zealand. Weka is an open-source software distributed for download under the GNU General Public License (GPL), making it freely accessible for users. The Weka workbench encompasses a diverse array of algorithms for data processing, predictive modeling, and visualization, complemented by user-friendly graphical interfaces for efficient access to these functionalities. Weka empowers practitioners to train and predict machine learning models without the need for coding, offering a user-friendly General User Interface with a rich selection of state-of-the-art algorithms. Moreover, it provides practitioners with the flexibility to utilize various hyperparameter optimization approaches, including GridSearch and CVParameter Selection.

Weka offers a diverse array of ML models for data exploration, including pre-installed models like Support Vector Machine, Naive Bayes, Neural Networks, and Decision Trees. Additionally, the tool provides pre-installed EL models such as Voting, Stacking, and AdaBoost. One notable feature is the built-in tool for testing models while optimizing hyperparameters. This tool allows users to set minimum and maximum values for each parameter, along with the step of increment, streamlining the process of fine-tuning models for optimal performance.

The tool also incorporates numerous evaluation measures for assessing machine learning models, encompassing but not limited to Accuracy, Kappa statistics, Mean Absolute Error, Precision, F-Measure, and ROC Area. The inclusion of a Confusion Matrix proves beneficial for machine learning practitioners, as it readily provides Specificity and Sensitivity values. Notably, all these functionalities are accessible through the graphical user interface (GUI) of the tool, eliminating the need for coding. This user-friendly approach proves particularly advantageous for practitioners without a background in Software Engineering, enabling them to effortlessly train and evaluate machine learning models without writing a single line of code. However, a significant drawback of employing Weka is the occasional extended time required for model evaluation. The duration is contingent on various factors, including the specific model used, the optimized hyperparameters, the number of features, and the volume of instances in the data undergoing exploration.

D. EVALUATION MEASURES

The efficiency and accuracy of the ML algorithms were assessed using a variety of metrics. Recognizing the complexity of assessing algorithm success with a single metric, we employed five: accuracy, Receiver Operating Characteristic (ROC), Kappa, Specificity, and Sensitivity.

1) ACCURACY

This metric refers to how accurately the algorithm classifies the production data. The formula for calculating accuracy based on correctly categorized records and total number of records is shown below. As a result, the better the classifier is, the higher the precision.

2) ROC

Using accuracy alone is not the best way to test an algorithm's performance; it is advised to use more than one measurement metric to get the whole story behind a dataset. Therefore, we can use the ROC curve to get a bigger picture. The ROC chart can be used to visually compare and contrast False Positive (FP) rate specificity and True Positive (TP) rate sensitivity. The specificity is represented on the x axis, while the sensitivity is represented on the y axis. The curve should come as close to the upper-left corner as possible. Figure 2 shows the ROC curve and the relationship between sensitivity and specificity.

3) KAPPA

Kappa measures an algorithm's success by comparing it to a classifier that makes predictions based on guessing at random. The higher and closer the Kappa to one, the better the algorithm.

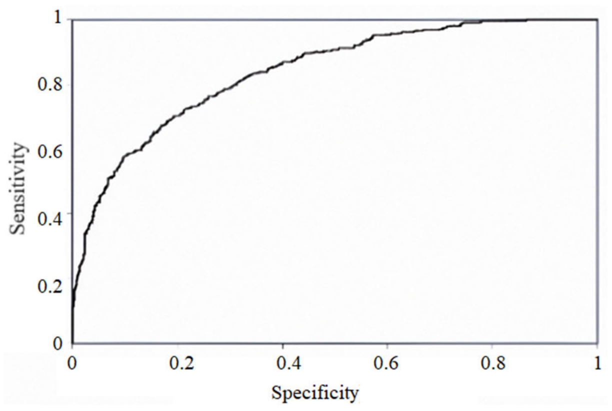

FIGURE 2. ROC curve.

4) SENSITIVITY

Sensitivity is described as the proportion of correctly expected positive examples. It is measured by dividing the number of TP impressions by TP value and adding the FN value [28].

$$TPR = \frac{TP}{FN + TP} \qquad (1)$$

5) SPECIFICITY

Specificity is the ratio of the negative examples correctly predicted. The specificity is calculated by dividing the TN on the FP plus TN [29].

$$FPR = \frac{FP}{TN + FP} \qquad (2)$$

6) F-MEASURE

F-measure is a commonly used metric for evaluating the success of classification techniques. To quantify this value, two measures called "Precision" and "Recall" must be computed, since the F-measure is the average of these two measures [29], [30], as shown below:

$$F - measure = \frac{precision * Recall * 2}{precision + Recall} \qquad (3)$$

V. RESULTS AND DISCUSSION

To address the research questions, we conducted a study to compare the results between single hypothesis ML models and EL models. The outcomes of these models were then thoroughly analyzed and compared. The evaluation involved assessing four EL models against five single hypothesis learning models. Table 3 shows the results of all nine tests based on the highest accuracy and performance of the ML models. The WEKA program was instrumental in running these models, employing a 10-fold cross-validation method to iteratively divide instances for training and production ten times.

A. RQ1: DO EL MODELS SHOW BETTER PERFORMANCE COMPARED TO SINGLE LEARNING MODELS IN SOFTWARE BUG PREDICTION PROBLEMS?

According to the conducted study and the results presented in Table 3, it is evident that EL models exhibit higher

TABLE 3. Performance metrics of machine learning algorithms.

Algorithm	Accuracy	ROC	Kappa	Sensitivity	Specificity	F-measure
Vote	0.7189	0.763	0.1898	0.828	0.373	0.702
Bagging	0.7178	0.749	0.2325	0.836	0.411	0.782
Random Forest	0.7172	0.716	0.2366	0.839	0.425	0.715
AdaBoostM1	0.7169	0.781	0.2189	0.809	0.416	0.766
Logistics Regression	0.5135	0.613	0.1395	0.722	0.406	0.759
SVM	0.6107	0.522	0.0664	0.713	0.359	0.639
Neural Networks	0.5094	0.519	0.0683	0.713	0.318	0.640
Naive Bayes	0.6042	0.509	0.1932	0.632	0.314	0.670
Decision Tree	0.5954	0.656	0.1901	0.733	0.357	0.667

TABLE 4. Results of ensemble learning models before and after hyperparameter optimization.

Algorithm	Accuracy	ROC	Kappa	Sensitivity	Specificity	F-measure
Vote	0.7189	0.763	0.1898	0.828	0.373	0.702
Optimized Vote	0.8195	0.763	0.1957	0.830	0.370	0.774
Percentage improvement	13.98%	no change	3.11%	0.24%	no change	10.26%
Bagging	0.7178	0.749	0.2325	0.836	0.411	0.782
Optimized Bagging	0.8183	0.769	0.2337	0.836	0.426	0.789
Percentage improvement	13.99%	2.67%	0.52%	no change	3.64%	0.90%
AdaBoostM1	0.7169	0.781	0.2189	0.809	0.416	0.766
Optimized AdaBoostM1	0.8176	0.797	0.2530	0.840	0.437	0.787
Percentage improvement	14.03%	2.05%	15.6%	3.83%	5.05%	2.74%
Random Forest	0.7172	0.716	0.2366	0.839	0.425	0.715
Optimized Random Forest	0.8182	0.757	0.2485	0.879	0.432	0.796
Percentage improvement	14.07%	5.72%	5.03%	4.77%	1.65%	11.33%

TABLE 5. A comparison of results achieved in our work with those in previous work.

Dataset	Study	Algorithm	Accuracy	ROC	F-measure
JM1		**Vote**			
	[17]		-	-	-
	[31]		-	-	-
	[32]		-	-	-
	[33]		0.759	-	0.326
	Our work		0.8195	0.763	0.774
		Bagging			
	[17]		-	0.697	0.754
	[31]		0.8118	-	-
	[32]		0.686	0.773	-
	[33]		0.732		0.327
	Our work		0.8183	0.769	0.783
		AdaBoostM1			
	[17]		-	0.700	0.727
	[31]		0.7912	-	-
	[32]		0.722	0.749	-
	[33]		0.700	-	0.373
	Our work		0.8176	0.797	0.787
		Random Forest			
	[17]		-	-	-
	[31]		0.7167	0.715	0.760
	[32]		-	-	-
	[33]		0.755	-	0.335
	Our work		0.8182	0.757	0.796

accuracy rates and superior performance in software bug prediction. The top-performing models among the tested ones are EL models, namely Vote, Bagging, Random Forest, and AdaBoosstM1, outperforming the single hypothesis learning models. These models, including Logistics Regression, SVM, Neural Networks, Naive Bayes, and Decision Trees, scored lower in terms of accuracy and ROC measures. Notably, the Vote EL model achieved the highest accuracy (0.7189). The ROC measure achieved for the same model was 0.763 which indicated good discrimination between positive and negative instances.

Furthermore, it is noteworthy that the ROC curve was consistently higher for the majority of EL models, approaching closer to one compared to single hypothesis learning models like Neural Networks, Naive Bayes, and Decision Trees.

The Kappa statistics also indicated higher values for the majority of EL models, with the exception of the Vote classifier model, which scored the highest in accuracy. In terms of sensitivity, EL models demonstrated an advantage, while the specificity was lower for EL models, with the Vote classifier registering the lowest value in this regard. Additionally, the F-measure scores for EL models surpassed those of single hypothesis learning models, signifying that EL models exhibited higher precision and recall compared to their single hypothesis learning counterpart.

B. RQ2: DOES ADDING HYPERPARAMETER OPTIMIZATION APPROACHES IMPROVE THE PERFORMANCE AND ACCURACY OF THE EL?

To address the second research question, hyperparameter optimization was conducted within the WEKA software, encompassing manual optimization for some hyperparameters and automated optimization using the CVParameter Selection Classifier, incorporating steps for each hyperparameter optimization to facilitate automated testing. As a result, testing time was increased significantly due to the fact that the model is tested on each step and combination of the hyperparameter, and the highest achieved combination is chosen to calculate the accuracy of the model. Table 3 shows the results of the impact of optimizing hyperparameters for EL models for software bug prediction.

The results presented in Table 4 clearly indicate that tuning hyperparameters for Ensemble Learning (EL) models significantly influences the accuracy and performance of the models in software bug prediction. Across all cases, there was an observable increase in all model measures, demonstrating the positive impact of hyperparameter optimization. Many hyperparameters were adjusted to enhance accuracy and performance, deviating from default settings. The evaluation of the proposed model highlights that employing EL models with hyperparameter optimization for software bug prediction yields superior performance compared to single hypothesis learning models. Furthermore, an enhancement in the ROC Curve is evident in three out of the four EL algorithms following hyperparameter optimization.

C. COMPARISON WITH PREVIOUS WORK

In this section, we conducted a comparative analysis of our findings with previous studies in the field of software bug prediction and hyperparameter optimization. We specifically focused on several EL classifiers that employed the same dataset as ours (JM1). The objective is to check whether our study represents an enhancement over existing research in this domain or not. The details of the results obtained are listed in Table 5. For AdaBoostM1, our work outperforms previous studies in terms of accuracy (0.8176), ROC (0.797), and F-measure (0.787), demonstrating improvements over the reported values in [17], [31], [32], [33], and [34]. Similar trends are observed for the Bagging algorithm, where our study exhibits higher accuracy (0.8183), ROC (0.769), and F-measure (0.783) compared to the referenced works. Moreover, the Vote algorithm in our work achieves a noteworthy accuracy of 0.8195, ROC of 0.763, and F-measure of 0.774, highlighting significant advancements over the provided references. The Random Forest algorithm also displays improved performance in our study, with an accuracy of 0.8182, ROC of 0.757, and F-measure of 0.796.

Overall, the results suggest that our proposed approach, which incorporates hyperparameter optimization, enhances the predictive capabilities of various classifiers across different metrics, surpassing prior studies in the same domain and dataset.

D. THREATS TO VALIDITY

Several potential threats to the validity of this study have to be acknowledged. Sampling bias may restrict the generalizability of results to datasets with different characteristics. The imbalanced distribution of defective instances (19.32%) could introduce bias, impacting relevance to diverse class distributions. Duplicated instances in the JM1 dataset (24.16%) may affect the model's learning process, potentially overestimating predictive capabilities. The effectiveness of hyperparameter optimization may be algorithm-dependent, limiting generalizability. Additionally, exploring different evaluation metrics to assess and compare classifiers could provide a more comprehensive understanding of model performance. Acknowledging and addressing these challenges is crucial for enhancing the transparency of our results.

VI. THEORETICAL AND PRACTICAL IMPLICATIONS

This study contributes theoretically by advancing our understanding of predictive modeling in software bug detection. By comparing ensemble learning (EL) models against single learning models, the research sheds light on the superiority of EL models in predicting software bugs. This analysis underscores the potential of EL models as a more effective approach in such tasks. Furthermore, the investigation into the impact of hyperparameter optimization on EL model performance offers insights into the importance of fine-tuning model parameters, enriching our theoretical understanding of machine learning model optimization.

On a practical level, the findings hold significance for software developers and quality assurance professionals. Demonstrating the superiority of EL models, particularly when optimized with hyperparameters, in predicting software bugs offers tangible benefits for software development processes. Implementing EL models with optimized hyperparameters can lead to more accurate bug prediction, ultimately improving software quality and reducing development costs. Additionally, the study provides practical guidance for selecting appropriate machine learning models for similar predictive tasks, aiding practitioners in making informed decisions.

Moreover, the introduction of various model evaluation measures such as accuracy, ROC Curve, Kappa statistics, Sensitivity, Specificity, and F-measure serves to enhance performance evaluation practices. These measures offer valuable tools for assessing the performance of machine learning models in software bug prediction, contributing to the development of standardized evaluation practices in the field. Such standardization enables more reliable performance comparisons across different studies and experiments, fostering advancements in software bug prediction methodologies.

VII. CONCLUSION AND FUTURE WORK

This study was conducted to enhance the effectiveness of software bug prediction by addressing two key questions. Firstly, we explored whether Ensemble Learning (EL) models outperform single learning models in predicting software bugs. Secondly, we investigated the impact of adding Hyperparameter Optimization approaches on EL model performance, examining whether this enhancement leads to improved accuracy. These questions served as the foundation of our exploration, guiding us to understand the nuances of predictive modeling in software bug detection. Our findings unequivocally demonstrate the superiority of EL models, especially those optimized with hyperparameters, over single hypothesis learning models in software bug prediction. Across various model measures, including accuracy, ROC Curve, Kappa statistics, Sensitivity, Specificity, and F-measure, EL models consistently outperformed their counterparts. The positive impact of hyperparameter tuning further reinforced the efficacy of EL models, contributing to enhanced accuracy and overall performance.

Nevertheless, recognizing a limitation in our study is crucial due to the reliance on a specific dataset from the NASA repository. To further validate the effectiveness of our proposed method, future work should consider incorporating more diverse datasets. This expansion will enhance the generalizability of our findings and ensure the robustness of our approach across various software development contexts. Additionally, we plan to extend our investigation by applying EL models with hyperparameter optimization to cross-project observations for software bug prediction. This aims to assess the generalizability of the proposed model and replicate its outcomes in diverse software bug prediction projects. Such cross-project validation will provide valuable insights into the robustness and applicability of our approach across different software development contexts. Finally, we plan to incorporate statistical tests to enhance the methodological approach, including parametric tests such as t-test and Wilcoxon, to compare the performance of classifiers. This addition will provide a more comprehensive assessment of the robustness and applicability of our approach in diverse software development contexts.

CONFLICT OF INTERESTS

The authors declare no conflict of interest.

DATA AVAILABILITY STATEMENT

The data presented in this study are available in the article.

REFERENCES

[1] R. Ferenc, D. Bán, T. Grósz, and T. Gyimóthy, "Deep learning in static, metric-based bug prediction," *Array*, vol. 6, Jul. 2020, Art. no. 100021.

[2] S. D. Immaculate, M. F. Begam, and M. Floramary, "Software bug prediction using supervised machine learning algorithms," in *Proc. Int. Conf. Data Sci. Commun. (IconDSC)*, Mar. 2019, pp. 1–7.

[3] S. Shalev-Shwartz and S. Ben-David, *Understanding Machine Learning: From Theory to Algorithms*. Cambridge, U.K.: Cambridge Univ. Press, 2014.

[4] F. Osisanwo, J. Akinsola, O. Awodele, J. Hinmikaiye, O. Olakanmi, and J. Akinjobi, "Supervised ML algorithms: Classification and comparison," *Int. J. Comput. Trends Technol.*, vol. 48, no. 3, pp. 128–138, 2017.

[5] A. Hammouri, M. Hammad, M. Alnabhan, and F. Alsarayrah, "Software bug prediction using ML approach," *Int. J. Adv. Comput. Sci. Appl.*, vol. 9, no. 2, pp. 78–83, 2018.

[6] D. Bowes, T. Hall, and J. Petrić, "Software defect prediction: Do different classifiers find the same defects?" *Softw. Quality J.*, vol. 26, no. 2, pp. 525–552, Jun. 2018.

[7] W. Fu, T. Menzies, and X. Shen, "Tuning for software analytics: Is it really necessary?" *Inf. Softw. Technol.*, vol. 76, pp. 135–146, Aug. 2016.

[8] H. Osman, M. Ghafari, and O. Nierstrasz, "Hyperparameter optimization to improve bug prediction accuracy," in *Proc. IEEE Workshop Mach. Learn. Techn. Softw. Quality Eval. (MaLTeSQuE)*, Feb. 2017, pp. 33–38.

[9] J.-C. Lévesque, C. Gagné, and R. Sabourin, "Bayesian hyperparameter optimization for ensemble learning," 2016, *arXiv:1605.06394*.

[10] T. Xia, R. Krishna, J. Chen, G. Mathew, X. Shen, and T. Menzies, "Hyperparameter optimization for effort estimation," 2018, *arXiv:1805.00336*.

[11] D. Di Nucci, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto, and A. De Lucia, "A developer centered bug prediction model," *IEEE Trans. Softw. Eng.*, vol. 44, no. 1, pp. 5–24, Jan. 2018.

[12] F. Khan, S. Kanwal, S. Alamri, and B. Mumtaz, "Hyper-parameter optimization of classifiers, using an artificial immune network and its application to software bug prediction," *IEEE Access*, vol. 8, pp. 20954–20964, 2020.

[13] X. Yang, D. Lo, X. Xia, and J. Sun, "TLEL: A two-layer ensemble learning approach for just-in-time defect prediction," *Inf. Softw. Technol.*, vol. 87, pp. 206–220, Jul. 2017.

[14] A. O. Balogun, A. O. Bajeh, V. A. Orie, and A. W. Yusuf-Asaju, "Software defect prediction using ensemble learning: An ANP based evaluation method," *FUOYE J. Eng. Technol.*, vol. 3, no. 2, pp. 1–14, Sep. 2018.

[15] S. Qiu, L. Lu, S. Jiang, and Y. Guo, "An investigation of imbalanced ensemble learning methods for cross-project defect prediction," *Int. J. Pattern Recognit. Artif. Intell.*, vol. 33, no. 12, Nov. 2019, Art. no. 1959037.

[16] A. Sayed and N. Ramadan, "Early prediction of software defect using ensemble learning: A comparative study," *Int. J. Comput. Appl.*, vol. 179, no. 46, pp. 29–40, Jun. 2018.

[17] S. K. Pandey, R. B. Mishra, and A. K. Tripathi, "BPDET: An effective software bug prediction model using deep representation and ensemble learning techniques," *Expert Syst. Appl.*, vol. 144, Apr. 2020, Art. no. 113085.

[18] M. Claesen, J. Simm, D. Popovic, and B. Moor, "Hyperparameter tuning in Python using optunity," in *Proc. Int. Workshop Tech. Comput. ML Math. Eng.*, vol. 1, 2014, p. 3.

[19] Y. Jung, "Efficient tuning parameter selection by cross-validated score in high dimensional models," *Int. J. Math., Comput., Phys., Elect. Comput. Eng., World Acad. Sci., Eng. Technol.*, vol. 10, no. 1, pp. 19–25, 2016.

[20] T. G. Dietterich, "Ensemble learning," in *The Handbook of Brain Theory and Neural Networks*, vol. 2. Cambridge, U.K.: The MIT Press, 2002, pp. 110–125.

[21] Y. Freund and R. E. Schapire, "Experiments with a new boosting algorithm," in *Proc. 13th Int. Conf. Mach. Learn.*, 1996, pp. 148–156.

[22] Z.-H. Zhou, "Ensemble learning," in *Machine Learning*. Singapore: Springer, 2021, pp. 181–210.

[23] L. Breiman, "Bagging predictors," *Mach. Learn.*, vol. 24, no. 2, pp. 123–140, Aug. 1996.

[24] R. Polikar, "Ensemble learning," in *Ensemble Machine Learning*. Cham, Switzerland: Springer, 2012, pp. 1–34.

[25] G. Brown, "Ensemble learning," in *Encyclopedia of Machine Learning*. Berlin, Germany: Springer, 2010, pp. 393–402.

[26] D. H. Wolpert, "Stacked generalization," *Neural Netw.*, vol. 5, no. 2, pp. 241–259, Jan. 1992.

[27] T. M. Oshiro, P. S. Perez, and J. A. Baranauskas, "How many trees in a random forest?" in *Proc. 8th Int. Workshop ML Data Mining Pattern Recognit. (MLDM)*. Berlin, Germany: Springer, Jul. 2012, pp. 154–168.

[28] A. E. Mohamed, "Comparative study of four supervised ML techniques for classification," *Inf. J. Appl. Sci. Technol.*, vol. 7, no. 2, pp. 1–15, 2017.

[29] M. G. Al-Obeidallah, D. G. Al-Fraihat, A. M. Khasawneh, A. M. Saleh, and H. Addous, "Empirical investigation of the impact of the adapter design pattern on software maintainability," in *Proc. Int. Conf. Inf. Technol. (ICIT)*, Jul. 2021, pp. 206–211.

[30] A.-R. Al-Ghuwairi, D. Al-Fraihat, Y. Sharrab, H. Alrashidi, N. Almujally, A. Kittaneh, and A. Ali, "Visualizing software refactoring using radar charts," *Sci. Rep.*, vol. 13, no. 1, p. 19530, Nov. 2023, doi: 10.1038/s41598-023-44281-6.

[31] H. Alsawalqah, H. Faris, I. Aljarah, L. Alnemer, and N. Alhindawi, "Hybrid SMOTE-ensemble approach for software defect prediction," in *Software Engineering Trends and Techniques in Intelligent Systems*. Cham, Switzerland: Springer, 2017, pp. 355–366.

[32] S. A. El-Shorbagy, W. M. El-Gammal, and W. M. Abdelmoez, "Using SMOTE and heterogeneous stacking in ensemble learning for software defect prediction," in *Proc. 7th Int. Conf. Softw. Inf. Eng.*, May 2018, pp. 44–47.

[33] D. Al-Fraihat, Y. Sharrab, F. Alzyoud, A. Qahmash, M. Tarawneh, and A. Maaita, "Speech recognition utilizing deep learning: A systematic review of the latest developments," *Hum.-Centric Comput. Inf. Sci.*, vol. 14, pp. 1–34, Mar. 2024.

[34] R. Li, L. Zhou, S. Zhang, H. Liu, X. Huang, and Z. Sun, "Software defect prediction based on ensemble learning," in *Proc. 2nd Int. Conf. Data Sci. Inf. Technol.*, Jul. 2019, pp. 1–6.

DIMAH AL-FRAIHAT received the Ph.D. degree in software engineering from the University of Warwick, U.K. She is currently an Assistant Professor with the Faculty of Information Technology, Isra University, Jordan. Her research interests include software engineering, refactoring, design patterns, software testing, requirements engineering, documentation, computer-based applications, technology enhanced learning, and deep learning.

YOUSEF SHARRAB received the Ph.D. degree in computer engineering from Wayne State University, Detroit, MI, USA, in 2017. He is currently an Assistant Professor with the Department of Computer Science, Isra University. His main research interests include artificial intelligence, machine learning, deep learning, computer vision, software engineering, and the IoT.

ABDEL-RAHMAN AL-GHUWAIRI received the Ph.D. degree in software engineering from New Mexico State University, Las Cruces, NM, USA, in 2013. He is currently an Associate Professor with the Department of Software Engineering, Hashemite University, Jordan. His research interests include software engineering, cloud computing, requirements engineering, information retrieval, big data, and database systems.

HAMZEH ALSHISHANI received the M.Sc. degree in software engineering from Hashemite University, Jordan. He is currently a Lecturer and a Researcher with the Department of Software Engineering, Hashemite University. His research interests include software engineering, cloud computing, and requirements engineering.

ABDULMOHSEN ALGARNI received the Ph.D. degree from the Queensland University of Technology, Australia, in 2012. He was a Research Associate with the School of Electrical Engineering and Computer Science, Queensland University of Technology, in 2012. He is currently an Associate Professor with the College of Computer Science, King Khalid University. His research interests include artificial intelligence, data mining, text mining, machine learning, information retrieval, and information filtering.