JITBoost: Boosting Just-In-Time Defect Prediction Using Boolean Combination of Classifiers

Mohammed A. Shehab[1], Abdelwahab Hamou-Lhadj[1], and Venkata Sai Gunda[2]
[1]Concordia University, Montreal, QC, Canada
[2]IIT Kharagpur, Kharagpur, India
m_shehab@live.concordia.ca, wahab.hamou-lhadj@concordia.ca, gundavenkatasai53@gmail.com

***Abstract*—Just-In-Time Software Defects Prediction (JIT-SDP) plays a critical role in software engineering by enabling the early identification of potential defects before they impact system performance. This study investigates the effectiveness of Boolean Combination of Classifiers (BCC) in building effective JIT-SDP models. We propose the JITBoost framework, which leverages three BCC algorithms, namely Brute-force Boolean Combination (BBC), Iterative Boolean Combination (IBC), and Weighted Pruning Iterative Boolean Combination (WPIBC). JITBoost combines the decisions of six traditional machine learning algorithms and one deep learning algorithm. When applied to 259K commits of 34 projects, we show that JITBoost models perform better than traditional machine learning and deep learning algorithms when used individually. Specifically, JITBoost-BBC, JITBoost-IBC, and JITBoost-WPIBC achieve mean AUCs of 0.891, 0.879, and 0.886, respectively, with cross-validation. With a time-aware data-splitting approach, they achieve mean AUCs of 0.863, 0.854, and 0.857, respectively. Overall, the findings suggest that combining machine learning models within the JITBoost framework can lead to improved performance in JIT-SDP models.**

***Index Terms*—Just-In-Time Software Defects Prediction, Machine Learning, Deep Learning, Boolean Combination of Classifiers, Software Reliability.**

I. Introduction

Identifying software defects in a timely manner is vital for ensuring optimal system performance and reliability [1]–[3]. Just-In-Time Software Defects Prediction (JIT-SDP) models have emerged as a promising approach in the field of software engineering [1], [3]. These models employ Machine Learning (ML) [4]–[6] and Deep Learning (DL) [7]–[9] techniques to build models using normal and buggy code commits in order to predict bugs at the code commit level, enabling developers to identify problematic code changes before they reach the central code repository [1], [3].

By integrating JIT-SDP models with continuous code quality tools, developers receive immediate feedback on potentially flawed code [1]. This allows them to address and rectify issues before they are incorporated into the main code base. As a result, the adoption of JIT-SDP models reduces the cost and effort associated with software quality assurance practices [10]. Furthermore, by providing timely insights into defect-prone areas, these models empower developers to prioritize testing efforts, allocate resources efficiently, and enhance the overall software quality [3].

In this study, we propose a framework, called JITBoost, which uses Boolean Combination of Classifiers (BCC) [11] for the prediction of buggy commits. BCC leverages Boolean functions to create classifiers that combine the decisions of individual classifiers with the aim to improve the overall prediction accuracy. Specifically, we investigate the use of three BCC algorithms (discussed in more details in Section III): Brute-force Boolean Combination (BBC) [11], Iterative Boolean Combination (IBC) [12], and Weighted Pruning Iterative Boolean Combination (WPIBC) [13]. These algorithms are used in the field of anomaly detection (e.g., [13] [12]) and have shown to perform better than single classifiers. We propose JITBoost-BBC, JITBoost-IBC, and JITBoost-WPIBC and compare their performance with individual JIT-SDP algorithms.

To evaluate the performance of the JITBoost framework, we conduct experiments using a dataset comprising 34 projects and a total of 259k commits. Our objective is to compare the effectiveness of JITBoost-BBC, JIT-Boost-IBC, and JIT-Boost-WPIBC algorithms against existing JIT-SDP techniques, which use individual traditional ML methods as well as DL algorithms.

Our research aims to address the following three research questions:

- **RQ1**: How does the performance of JITBoost algorithms compare to JIT-SDP models that use traditional machine learning algorithms?
- **RQ2**: How does the performance of JITBoost algorithms compare to a deep learning JIT-SDP algorithm?
- **RQ3**: How does the combination of traditional JIT-SDP models and deep learning models affect the performance of the JITBoost algorithms?

For RQ1, we combine the decisions of six traditional classifiers (see Section III) using BBC, IBC, and WPIBC and compare the results with that of each individual classifier. We found that all Boolean combination algorithms perform better than the single ML algorithms. For RQ2, we compare the Boolean combination classifiers of RQ1 with the newly proposed JIT-SDP DL algorithm DeepJIT [9]. We found that the combination of traditional ML algorithms performs better than when using DeepJIT. In the last question, RQ3, we compare the combination of six traditional classifiers and another combination of the same classifiers and DeepJIT. The objective is to see if the DL algorithm improves the accuracy of the combination. Our findings show that the accuracy is

only improved when using JITBoost-WPIBC.

These findings have important implications for practitioners in software development, as they suggest that simpler ML models may be just as effective as more complex DL models. This is also inline with the recent finding of Zeng et al. [9], which showed a simple ML method can outperform CC2Vec [8] and DeepJIT [7] when applied to very large datasets.

This study can benefit researchers and practitioners by proposing a JIT-SDP framework that can improve the accuracy of predicting buggy commits, leading to more reliable and accurate tools for defect prediction at the commit level.

The structure of the paper is as follows: The next section provides a review of software defect prediction techniques using ML and DL. In Section III, we present the Boolean combination algorithms. Section IV describes the study setup including the datasets, features, evaluation metrics, and the algorithms. In Section V, we present the results that address the research questions. Section VI outlines potential threats to validity and the actions taken to mitigate them. Finally, the paper concludes with Section VIII, which presents the conclusions and highlights future research directions.

II. Related Work

Code changes in software development can introduce bugs and costly mistakes. Researchers have used ML and DL techniques to build predictive models that analyze code changes and identify potential issues. Traditional metrics-based features have limitations in capturing the true meaning of code changes. To address this, Hoang et al. [7] proposed DeepJIT, a Deep Learning Framework for JIT-SDP. DeepJIT combines metrics-based on syntactic and semantic features to improve defect prediction accuracy. These features extracted from Commit Message (CM) and Code Changes (CC). Hoang et al. [7] tested DeepJIT on the QT and OpenStack projects, achieving the best AUC values of 0.788 and 0.814, respectively. They used different data splitting methods (cross-validation, long periods, short periods) but found no significant differences in the results with data splitting approaches.

In addition to DeepJIT, Hoang et al. [8] also proposed a CC2Vec framework for deep JIT-SDP. Like DeepJIT, CC2Vec uses natural language processing techniques to extract syntactic and semantic features from the commit message's and code changes. In addition to the these features, two vectors are created from the added and deleted lines in the commit as additional features. These two vectors are then used to enhance the classification process, a step known as Hierarchical Attention Network (HAN). HAN is utilized for training the Neural Tensor Network, a form of deep learning architecture. To achieve the best results for CC2Vec, six parameters must be tuned. Finally, the output of CC2Vec is used as input features for JIT-SDP, such as SVM. The proposed method increases AUC by 4%. Note, the CC2Vec is not end-to-end deep learning framework such as DeepJIT.

However, Pornprasit and Tantithamthavorn [4] proposed JITLine as an alternative approach for predicting risky code changes and identifying buggy code. In comparison to two existing models (DeepJIT [7] and CC2Vec [8]), JITLine achieved the highest AUC at 82%. JITLine differs from DeepJIT and CC2Vec models in that it employs ML algorithms that are less complex and more computationally efficient. Rather than relying on deep learning techniques, JITLine provides faster and simpler ML models for building JIT-SDP, such as those used in DeepJIT and CC2Vec. These results suggest that JITLine may be a promising alternative to more complex models for predicting risky code changes and identifying potential bugs.

Zeng et al. [9] explored the effectiveness and limitations of modern deep JIT-SDP techniques, specifically DeepJIT and CC2Vec. The authors created a larger dataset with over 310k commits and used it to evaluate the performance of DeepJIT, CC2Vec, and two traditional JIT-SDP methods. They discovered that CC2Vec does not consistently outperform DeepJIT, and neither technique can be guaranteed to outperform traditional JIT-SDP methods. Also, they found that all studied ML and DL JIT-SDP methods experienced a decrease in performance during cross-project validation and that increasing the training data size did not necessarily enhance prediction accuracy. Interestingly, Zeng et al. [9] demonstrated that a straightforward JIT-SDP approach (e.g., LA predict) which employs a traditional classifier with the added-line-number feature, could already surpass CC2Vec and DeepJIT in terms of both effectiveness and efficiency for most projects.

A semi-supervised learning approach, called Effort-Aware Training (EATT), has been proposed by Li et al. [14] to address the challenges of labeling data and reducing its noise. EATT uses a small amount of labeled data and the same features and projects as Kamei et al. [3]. P_{opt} and AUC metrics were used to evaluate the EATT model, as well as the supervised and unsupervised models, with EATT achieving an average of 0.73 AUC and 0.89 P_{opt}, while the supervised and unsupervised models achieved lower average scores.

Nayrolles and Hamou-Lhadj [1] proposed a JIT-SDP approach called CLEVER that combines commits from video game systems built on the same game engines. This training model uses clone detection techniques to make predictions, and extracts the corresponding code block for each suspected buggy commit, comparing it to a database of known defects. However, CLEVER may be too dependent on how Ubisoft systems are developed, as specific code segments may be reused across systems.

Lomio et al. [15] conducted a study to compare different approaches for detecting anomalies in Fine-Grained Just-in-Time Defect Prediction models. These models label commits into three classes proposed by Pascarella et al. [16], where the commits are labeled into three main classes (risky, partial-risky, and normal) rather than (risky and normal). The researchers used a modified SZZ algorithm to label the dataset. Then they compared three One-Class Classification (OCC) models with three binary classifiers and found that the OCC models performed better on imbalanced datasets, but not as well as binary classifiers in terms of AUC. The best performing

binary model was Extremely Randomized Trees with an AUC mean of 71%, followed by k-NN with an AUC mean of 60%, and Support SVM with the worst results at 31% AUC. The OCC models achieved a mean AUC of 50%. Lomio et al. [15] approach used at file level rather than commit level.

Shehab et al. [6] came up with approach to JIT-SDP with their ClusterCommit model. This model brings together commit data from various projects, grouping them into a larger cluster using a community detection algorithm. By building the JIT-SDP model using all project data of the cluster and evaluating each project separately, the ClusterCommit model achieved an impressive F1 score of 73% and 0.44 MCC. Using the same 14 code metrics proposed by Kamei et al. [3], ClusterCommit is a promising JIT-SDP model that could revolutionize the way we approach defect prediction.

III. Boolean Combination of Classifiers

The Boolean combination of classifiers is an approach that uses Boolean logic operators, such as AND, OR, and NOT, to combine the decision of multiple classifiers into a single classifier. These classifiers can be of any type, including decision trees, Support Vector Machines, logistic regression, etc. The approach works by first generating a set of classifier predictions based on the available features in the dataset. Then, it combines these predictions on the Receiver Operator Characteristics (ROC) curve space using Boolean operators to form the final new classifier.

Consider the decision vectors of two classifiers A and B. We can combine the decisions using six Boolean operators: $\wedge$, $\vee$, $\neg$, $\oplus$, $\neg\wedge$, $\neg\vee$, $\equiv$. There exist 10 possible ways to combine these decisions, namely $A \wedge B$, $\neg A \wedge B$, $A \wedge \neg B$, $\neg(A \wedge B)$, $A \vee B$, $\neg A \vee B$, $A \vee \neg B$, $\neg(A \vee B)$, $A \oplus B$, and $A \equiv B$. Each of these combinations results in a new classifier, which may or may not improve the accuracy of the individual classifiers. The idea of Boolean combination of classifiers is to explore the space of all possible combinations in order to find the combination that provide best accuracy on the ROC curve [11]. Doing so, however, may result in computational overhead as the number of classifiers increase. In this study, we experiment with three different Boolean combination algorithms, namely Brute-force Boolean Combination (BBC) [11], Iterative Boolean Combination (IBC) [17], and Weighted Pruning Iterative Boolean Combination (WPIBC) [13].

Figure 1 illustrates an example of two models, A and B, in the context of Boolean combination of classifiers. The dashed line in the plot represents the result of combining the two models using a Boolean function. Each point on the dashed line corresponds to the combination of a point from model A and another point from model B. The BBC algorithm explores all possible Boolean combinations to plot the dashed line in the ROC space. This approach allows for optimizing the performance of the combined classifier and finding the best trade-offs between true positive and false positive rates.

1) Brute-force Boolean Combination (BBC): The Brute-force Boolean Combination algorithm is an exhaustive search algorithm that generates all possible combinations of Boolean operators. This approach tests all combinations of the individual classifier outputs to find the combination with the highest classification accuracy [18].

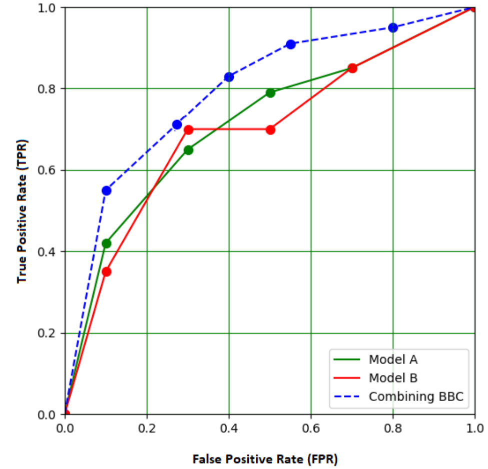

Fig. 1: Example of combining two models in the ROC space

Suppose we have three individual classifiers, each classifier produces a binary output (either 0 or 1) for a given input. The BBC approach explores all $2^3 = 8$ possible combinations of the three binary outputs (000, 001, 010, 011, 100, 101, 110, 111) to see which combination results in the highest classification accuracy. Boolean logic operators are applied to the outputs (predictions) to create a final prediction. Then evaluate the classification accuracy of each combination by comparing the predicted output to the true output for a set of test labels [18].

The combinations produce the highest classification accuracy to be selected as the optimal combination for the given classifiers. However, this approach can become computationally expensive as the number of individual classifiers increases since the number of possible combinations grows exponentially with the number of classifiers [18].

2) Iterative Boolean Combination (IBC): Iterative Boolean Combination (IBC) is another approach for combining multiple classifiers into a single classifier using Boolean operators. Unlike the BBC approach that tries all possible combinations of classifiers, IBC combines classifiers iteratively until a satisfactory level of accuracy is achieved [19].

The IBC algorithm operates by first selecting an initial subset of classifiers, such as the top-performing classifiers in terms of AUC or accuracy. Then, IBC iteratively combines this subset of classifiers using Boolean operators (such as AND, OR, and NOT) to generate a new complex classifier. The performance of the new classifier is evaluated using a validation set, and the process is repeated until a satisfactory level of performance is achieved [19].

The main difference between IBC and BBC is that IBC is more efficient as it does not try all possible combinations of classifiers. Instead, it starts with an initial subset of classifiers and iteratively combines them until a satisfactory level of

performance is achieved. Suppose we want to synthesize a boolean function that satisfies the following properties: 1) It has three input variables (A, B, and C). 2) It outputs 1 if and only if exactly two of its inputs are 1. Using the BBC method, we would need to consider all possible combinations of the input variables ($2^3 = 8$ combinations) and evaluate the output of the function for each combination. We could then use these evaluations to construct a truth table and derive the boolean expression that satisfies the desired properties. This approach can be time-consuming and impractical for larger functions with many input variables [11], [17]. On the other hand, the IBC algorithm could be used to synthesize the function more efficiently [19]. We can start with a set of initial functions that satisfy some of the desired properties, such as: $(A \wedge B, A \wedge B, B \wedge C)$. We can then iteratively combine these functions to generate larger functions that satisfy more of the desired properties. For example, we can combine the first two functions using the OR operator to obtain: $(A \wedge B \wedge \neg C) \vee (A \wedge \neg B \wedge C) \vee (\neg A \wedge B \wedge C)$. This function satisfies two of the desired properties: it outputs 1 if and only if exactly two of its inputs are 1, and it outputs 0 if all inputs are 0.

This function satisfies all of the desired properties and can be expressed using only three Boolean operators. The IBC algorithm is able to synthesize this function much more efficiently than the BBC, which would have required evaluating all possible input combinations [19].

3) Weighted Pruning Iterative Boolean Combination (WPIBC): Weighted Pruning Iterative Boolean Combination (WPIBC) is an extension of IBC that aims to improve the performance of the model. WPIBC uses a weighted kappa score. The weighted kappa score takes into account the similarity between the predictions of the classifiers, as well as the degree of difficulty in making the prediction. The kappa score is measured using Equation 1.

$$kp = \frac{2 * (TP * TN - FN * FP)}{(TP + FP) * (FP + TN) * (TP + FN) * (FN + TN)} \quad (1)$$

The WPIBC algorithm starts by generating an initial set of classifiers using IBC. Then, for each classifier in the ensemble, the weighted kappa score is calculated using the similarity predictions. The classifiers with the similar weighted kappa scores are pruned from the ensemble steps, then the process is repeated iteratively until no more classifiers can be pruned. The remaining classifiers are then combined using boolean operators to generate the final classifier. The main advantage of WPIBC over previous algorithms is its ability to identify and remove classifiers that are not contributing to the performance of the ensemble. By using the weighted kappa score as a pruning metric, WPIBC can identify classifiers that are making poor predictions and remove them from the ensemble, improving the overall performance of the model [13].

In comparison, the BBC algorithm generates all possible combinations of classifiers, which can lead to a large number of classifiers and computational complexity. On the other hand, IBC generates classifiers iteratively, which can be computationally efficient but may not identify and remove poorly performing classifiers [13]. WPIBC combines the benefits of both approaches by generating classifiers iteratively while also identifying and removing poorly performing classifiers using the weighted kappa score, resulting in a more effective and efficient ensemble model [13].

IV. Study Setup

This section represents the overall configuration for our study. First, we present the dataset description and feature extraction. Next, we discuss the data labeling and splitting approaches. After that, the evaluation metrics that are used to measure accuracy. Then, the algorithms used to build JITBoost are presented.

A. Datasets Description and Features Extraction

To assess the effectiveness of our approach, we conducted a study on 34 open-source projects from the Apache Foundation. Our focus was on projects written in Java programming language, and we only considered projects with a minimum of 1,000 commits to ensure sufficient historical data for predictive purposes. These projects are available on GitHub and operate Jira as a bug-tracking system. A comprehensive overview of the dataset is provided in Table I, which includes project names, normal and buggy commit counts, following by defects ratio, with a total of commits. The total size of dataset is 259k of commits.

We performed feature extraction from each project in Table I using the GIT version control system. We extract 14 features proposed by Kamei et al. [3], which are widely used in the JIT-SDP area (e.g., [20] [9] [21] [15]), were extracted alongside 2 additional features, Code Change (CC) and Code Message (CM), proposed by Hoang et al. [7]. These features capture semantic information and syntactic structure, both hidden within the source code, and were used to construct and evaluate the DeepJIT model. Table II presents the 16 features we extracted from the projects. All the data used in this paper is made available online[1].

B. Data Labeling

To label the commits into normal and buggy commits, we used the Refactoring Aware SZZ Implementation (RA-SZZ) algorithm, introduced by Campos Neto et al. [22]. This algorithm extracts bug reports and the corresponding code changes from a version control system, such as GIT, to identify the code changes related to the bug. It determines the code version where the bug originated, identifies the source code changes between that version and the fix, and detects any code refactorings performed. The RA-SZZ algorithm improves bug localization accuracy by considering code refactoring effects, reducing false positives compared to the original SZZ algorithm [23].

[1]https://doi.org/10.5281/zenodo.8206280

TABLE I: Description of the Datasets

Project Name	Normal	Buggy	Defects %	Total
Accumulo	9,541	552	5.5%	10,093
Airavata	6,729	497	6.9%	7,226
Ambari	24,477	110	0.4%	24,587
Avro	2,151	235	9.8%	2,386
Bigtop	2,567	31	1.2%	2,598
Bookkeeper	2,289	84	3.5%	2,373
Camel	9,032	3,990	30.6%	13,022
Carbondata	4,249	552	11.5%	4,801
Cayenne	6,365	285	4.3%	6,650
Cocoon	13,094	66	0.5%	13,160
Curator	2,690	28	1.0%	2,718
Derby	7,795	473	5.7%	8,268
Drill	2,288	1,643	41.8%	3,931
Falcon	2,096	130	5.8%	2,226
Flink	20,369	4,613	18.5%	24,982
Flume	1,151	661	36.5%	1,812
Gora	1,314	52	3.8%	1,366
Hadoop	9,881	627	6.0%	10,508
Hbase	16,721	1,058	6.0%	17,779
Helix	3,672	56	1.5%	3,728
Hive	11,759	518	4.2%	12,277
Ignite	13,969	1,609	10.3%	15,578
Jackrabbit	8,488	370	4.2%	8,858
Oodt	2,006	85	4.1%	2,091
Oozie	2,244	114	4.8%	2,358
Openjpa	3,404	1,706	33.4%	5,110
Parquet-mr	2,126	114	5.1%	2,240
Phoenix	3,284	168	4.9%	3,452
Reef	3,813	60	1.5%	3,873
Spark	19,591	376	1.9%	19,967
Storm	10,178	239	2.3%	10,417
Tez	2,426	232	8.7%	2,658
Zeppelin	4,259	543	11.3%	4,802
Zookeeper	1,453	577	28.4%	2,030
Total	***237,471***	***22,454***	**-**	***259,925***

TABLE II: The features used to build the JIT-SDP models.

Dimension	Name	Description
Diffusion	NS	Number of modified sub-systems
	ND	Number of modified directories
	NF	Number of modified files
	Entropy	Distribution of modified code across files
Size	LA	Added lines
	LD	Deleted lines
	LT	Line of code before edit
Purpose of Change	Fix	Whether or not the change is a defect or fix
History	NDEV	Number of developers that changed the file
	AGE	The average time between file changes
	NUC	The number of unique changes
Experience	EXP	Developer experience
	REXP	Recent developer experience
	SEXP	Developer experience on sub-systems
Commits	CC	Code changes inside commit
	CM	Commit message represented by the developer

C. Data Splitting and Preparation

We used two approaches, Cross-Validation (CV) and Time-aware Validation (TV), for training and evaluating the models [7]. In CV, the dataset is divided into a training set (70%) and a testing set (30%), with the testing set kept hidden during model training. This ratio was chosen to compare the data sizes between CV and TV approaches. It was observed that the buggy commits in the dataset occurred in the last 30% of commits based on sorting by commit time (Figure 2) [9].

For this research, both CV and TV approaches were employed, following the methodology of Zeng et al. [9]. We used ten-fold CV to hyper-tune the model parameters, where nine folds were used for training and one fold for validation. These steps are done only with the training set (70%). This procedure was repeated ten times, as suggested by Zeng et al. [9], to ensure fair testing, and the average of the ten tests was recorded as the CV output. The best model was then evaluated using the previously hidden (30%) testing set. The use of CV is recommended to build more reliable models, prevent overfitting, and enhance generalization to unseen data [24], [25]. In this study, the testing set was randomly selected without replacement and held out from the training process.

The CV method, as mentioned in Tan et al. [26], does not take into account the temporal aspect of commits when selecting samples for training and testing. To address this limitation, other JIT-SDP studies, such as McIntosh et al. [20], Hoang et al. [7], and Lomio et al. [15], have adopted a time sensitive (TV) approach for data splitting. The TV approach considers the time sensitivity of changes, where JIT-SDP models are trained using earlier data to predict buggy commits in later ones.

In our study, we followed a chronological sorting of the data and employed the TV approach for JIT-SDP models. The dataset was divided into training and testing sets using a split point determined by a 70% and 30% ratio, respectively, as shown in Figure 2. In addition to the TV approach, unlike previous studies such as McIntosh et al. [20], Hoang et al. [7], and Lomio et al. [15], we also performed a ten-fold CV approach using the early 70% of the data, similar to the approach conducted by Zeng et al. [9]. The number of repetitions in TV approach is one due to its restrictions. We applied the same setting as Hoang et al. [7] and Zeng et al. [9] to our dataset.

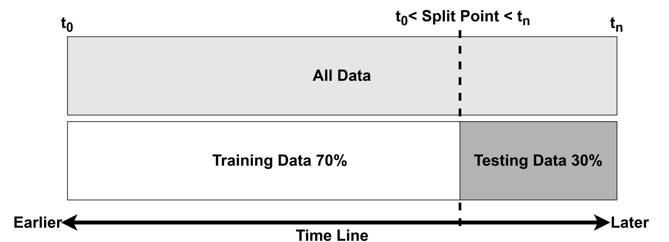

Fig. 2: Splitting dataset using time-aware validation

D. Evaluation Metrics

To evaluate the performance of classification models, we use the Area Under the ROC Curve (AUC) [27], [28]. The ROC curve is a graphical representation of the True Positive

Rate (TPR) against the False Positive Rate (FPR) at different decision thresholds. The TPR and FPR values used to create the ROC curve are calculated from the confusion matrix [29], which displays TP, TN, FP, and FN of a classification model based on the actual and predicted class labels of a dataset as shown in Table III.

TABLE III: Description of ROC curve and confusion matrix.

	Predicted Positive	**Predicted Negative**
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)

- True Positive (TP): The number of buggy commits that are correctly classified as buggy
- False Positive (FP): The number of normal commits, classified as buggy (a.k.a false alarms)
- False Negative (FN): The number of buggy commits that are classified as normal
- True Negative (TN): The number of normal commits that are correctly classified as normal

E. Algorithms

We propose three JITBoost models (JITBoost-BBC, JITBoost-IBC, JITBoost-WPIBC) that use three different BCC techniques to enhance the JIT-SDP accuracy. We use six traditional ML classification methods and one DL algorithm to create JITBoost (see Figure 3).

The traditional ML algorithms are: Naive Bayes (NB), Random Forest (RF), Decision Tree (DT), Support Vector Machine (SVM), Logistics Regression (LR), k-Nearest Neighbors (k-NN). We chose these algorithms because they are used extensively in the field of JIT-SDP (e.g., [1], [3]).

We used the end-to-end deep learning framework (DeepJIT) [9]. Unlike other approaches such as DBN-JIT [7] and CC2Vec [8], which only employ deep learning models to extract and build semantic information and syntactic structure from commit messages and code changes, DeepJIT takes a more comprehensive approach. DeepJIT not only utilizes a deep learning model for extracting semantic information and syntactic structure but also trains the model using a Convolutional Neural Network (CNN) algorithm [9]. Moreover, the DeepJIT is specifically selected for evaluation because it outperforms other DL models, such as DBN-JIT and CC2Vec, and its status as an end-to-end deep learning framework [9].

Figure 3 shows the process of combining these algorithms to create JITBoost models using BBC, IBC, and WPIBC. For RQ1 and RQ2, we combine the six traditional algorithms to create JITBoost algorithms. For RQ1, we compare the combined algorithms to each traditional ML algorithms. For RQ2, we compare the combination to Deep JIT. As for RQ3, we combine the six traditional ML algorithms and DeepJIT and compare that to a combination of only traditional ML.

V. Results Analysis and Discussions

In this section, we present and discuss the results of the experiments by providing answers to our research questions in the subsections.

A. RQ1: How does the performance of JITBoost algorithms compare to JIT-SDP models that use traditional machine learning algorithms?

Figure 4 shows the average AUC results of JITBoost models (JITBoost-BBC, JITBoost-IBC, JITBoost-WPIBC) and that of the six traditional ML methods (SVM, LR, etc.), using both CV and TV data splitting approaches. The results indicate that when using the CV approach, all JITBoost models perform better than the ML models. JITBoost-BBC, JITBoost-IBC, and JITBoost-WPIBC achieve average AUC values of 0.891, 0.879, and 0.886, respectively, while the best ML model (RF) achieves an average AUC of 0.857. When the TV data splitting approach is used, the performance of all ML models declines even further. The best ML model (RF) achieves an average AUC of 0.776, whereas all JITBoost algorithms still maintain higher average AUC (the worst result is 0.854 by JITBoost-IBC) results compared to the ML models.

We used the Mann-Whitney U test [30] [31] to determine the statistical significance of the model's results. The null hypothesis (h_0) assumes that the results of the models are not statistically different, while the alternative hypothesis (h_1) suggests that the model's results are statistically different. The null hypothesis is rejected when the p-value is less than 0.05 (95% confidence interval) [32].

Additionally, we used the Cliff's δ effect size to quantify the magnitude of the difference between the two groups. Cliff's δ ranges from -1 to 1, with 0 indicating no effect, negative values indicating Y values are greater than X, and positive values indicating X values are greater than Y. Different ranges are defined for interpreting the effect size: $0.147 < |\delta| \leq 0.33$ (small effect), $0.33 < |\delta| \leq 0.474$ (moderate effect), and $|\delta| > 0.474$ (large effect) [33], [34]. The calculation of Cliff's δ is based on Equation (2), where x and y represent two data vectors, and n_x and n_y denote the sizes of these vectors.

$$Cliff's\ \delta = \frac{\sum_i \sum_j sign(y_i - x_j)}{n_y . n_x} \tag{2}$$

To compare the performance based on the data splitting approach, the results using CV and TV are examined in Table IV. The table presents the average and standard deviation of AUC for all models, the improvement ratio (IM) measured as $(AUC_{CV} - AUC_{TV})/AUC_{CV}$, and the results of the Mann-Whitney U test and Cliff's δ. The findings indicate that JITBoost's performance is slightly improved by 3% with the TV approach compared to CV. However, the p-value suggests that the results of JITBoost models using CV and TV are not statistically different because p-value > 0.05, so we can not reject h_0, although a small effect size is observed.

On the other hand, the p-value of the ML models is less than 0.05, indicating statistically different results. The effect sizes vary, with RF and k-NN exhibiting large effects and NB, DT, LR, and SVM showing moderate effects. These results suggest that the data splitting approach (i.e., CV and TV) has a limited impact on the performance of JITBoost models, but it significantly affects the performance of ML models, with some models showing large or moderate effect sizes.

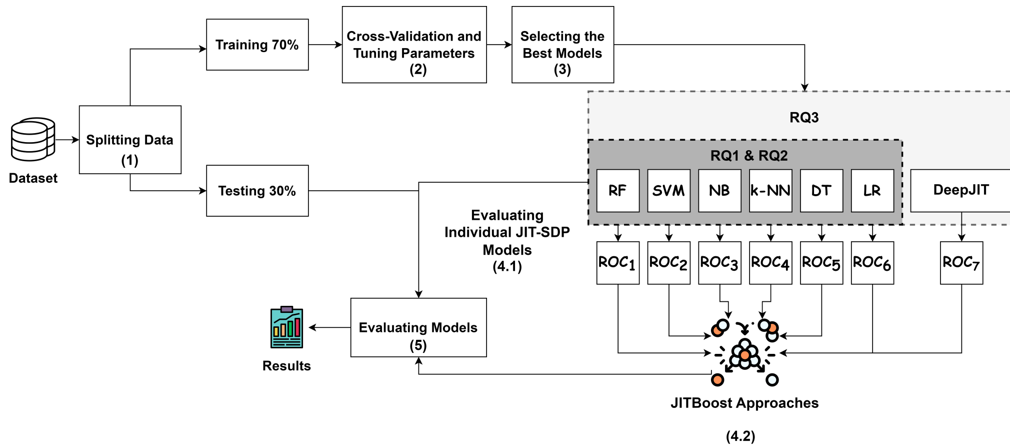

Fig. 3: The JITBoost Overall Approach

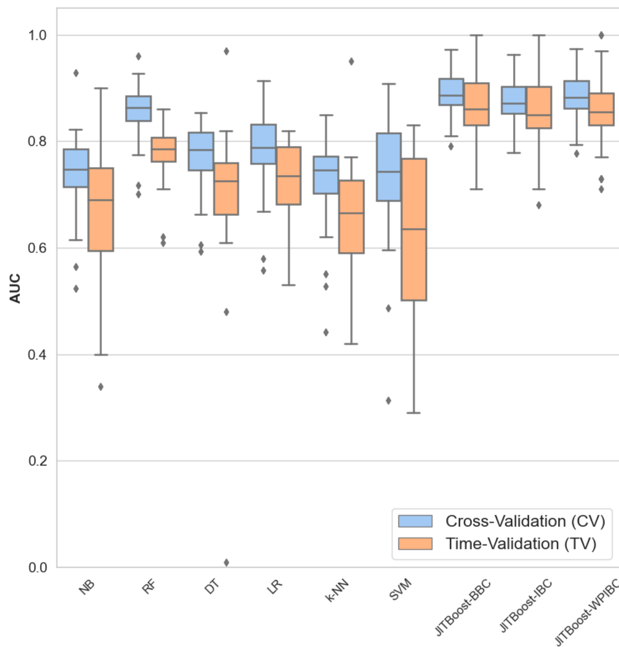

Fig. 4: Comparison of JITBoost models with ML models.

TABLE IV: The statistical analysis for models with different data splitting approaches (CV and TV).

	AUC_{CV} ($\mu \pm \sigma$)	AUC_{TV} ($\mu \pm \sigma$)	IM%	Cliff's δ	p-value
NB	0.742 ± 0.074	0.669 ± 0.136	10%	0.381	0.007
RF	0.857 ± 0.052	0.776 ± 0.056	10%	0.801	0.000
DT	0.767 ± 0.067	0.696 ± 0.146	9%	0.472	0.001
LR	0.783 ± 0.074	0.721 ± 0.082	8%	0.469	0.001
k-NN	0.724 ± 0.085	0.657 ± 0.102	9%	0.482	0.001
SVM	0.733 ± 0.118	0.626 ± 0.156	15%	0.392	0.006
DeepJIT	0.737 ± 0.101	0.774 ± 0.105	-5%	0.230	0.104
JITBoost-BBC	0.891 ± 0.041	0.863 ± 0.071	3%	0.267	0.058
JITBoost-IBC	0.879 ± 0.044	0.854 ± 0.077	3%	0.266	0.060
JITBoost-WPIBC	0.886 ± 0.045	0.857 ± 0.074	3%	0.262	0.063

Next, we examine the differences in performance based on the classifiers. Specifically, we compare each JITboost model individually to the other six models, using the same data splitting approach. The same comparison is performed for all other models used in this research question. Table V presents the performance of the models compared to others, using both the CV and TV approaches. All ML models perform lower than the JITBoost models. Furthermore, all p-values are less than 0.05, indicating statistically significant differences in results for CV and TV data splitting approaches. The effect size is moderate for models such as NB, RF, DT, and LR, while the SVM model a large effect size and small for k-NN with CV approach. Using TV approach, models (NB, DT, and k-NN) have small size effect, while RF and LR has moderate effect size and SVM a large effect size. On the other side, all JITBoost models have p-value less than 0.05, which also indicating to statistically differences with large effect size ($\delta > 0.474$). In summary, the results indicate that JITBoost-BBC models have statistically different results compared to ML models, with p-values less than 0.05 for CV and TV data splitting approaches.

TABLE V: Effect size by type of classifier

	CV		TV	
Classifier	**Cliff's δ**	**p-value**	**Cliff's δ**	**p-value**
NB	-0.408	0.000	-0.017	0.000
RF	-0.453	0.000	-0.413	0.000
DT	-0.325	0.002	-0.050	0.000
LR	-0.320	0.002	-0.341	0.001
k-NN	-0.197	0.009	-0.262	0.012
SVM	-0.524	0.000	-0.552	0.000
DeepJIT	-0.368	0.000	-0.015	0.000
JITBoost-BBC	0.577	0.000	0.531	0.000
JITBoost-IBC	0.473	0.000	0.477	0.000
JITBoost-WPIBC	0.529	0.000	0.494	0.000

Finding RQ1: Our findings show that JITBoost models outperform ML models by 18%, 17%, and 17% for JITBoost-BBC, JITBoost-IBC, and JITBoost-WPIBC, respectively, when using CV. Additionally, when using TV, JITBoost models show even a better performance with improvements of 36%, 35%, and 35% for JITBoost-BBC, JITBoost-IBC, and JITBoost-WPIBC, respectively, compared to ML models. While the choice of data splitting approach had small impact on JITBoost's performance, it had a significant effect on ML models, with improvements of at least of 8%.

B. RQ2: How does the performance of JITBoost algorithms compare to a deep learning JIT-SDP algorithm?

This research question compares JITBoost models to DeepJIT models. Figure 5 visualizes the results of JITBoost models created using six ML models only and a model created using DeppJIT. It can be seen that all JITBoost Models outperform DeepJIT. The JITBoost-BBC, JITBoost-IBC, and JITBoost-WPIBC achieve an average AUC of 0.890, 0.880, and 0.890, respectively, with the CV approach compared to an average AUC of 0.737 for DeepJIT. Also, they achieve an average AUC of 0.860, 0.850, and 0.860, respectively, with the TV approach, compared to 0.774 for DeepJIT. The JITBoost models not only achieve better results, but also result in lower variance (see the standard deviation in Table VI. In other words, JITBoost lead to a more confident prediction, reducing prediction errors that may be caused by overfitting [27].

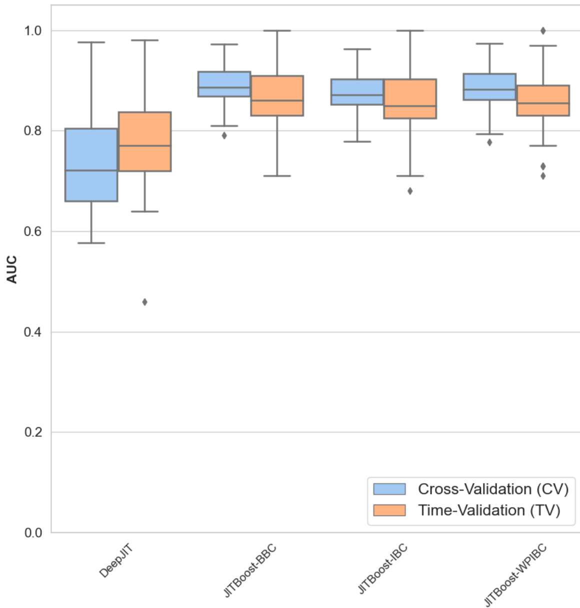

Fig. 5: Comparison between JITBoost models and DeepJIT models using both data splitting approaches CV and TV.

We used the Mann-Whitney U test and Cliff's δ size with the results of JITBoost and DeepJIT models. As shown in Table IV, DeepJIT using TV performs better than when using the CV data splitting approach (an improvement of 5%). However, the p-value of DeepJIT is greater than 0.05, so we can not reject the h_0. It means the results are not statistically different with small effect size. Therefore, we cannot claim that the DeepJIT model is affected by the data splitting approaches. This also applies to JITBoost models.

We statistically analyze the results of JITBoost models with DeepJIT models. Table V shows the effect size and p-value for each JITBoost model compared to DeepJIT. The p-value of all models is less than 0.05 with CV and TV data splitting approaches, meaning that there is a significant difference between JITBoost accuracy and that of DeepJIT. The effect size of DeepJIT is moderate using CV and small with TV.

Finding RQ2: Our findings show that JITBoost models exhibit superior performance compared to DeepJIT. Using CV, JITBoost-BBC, JITBoost-IBC, and JITBoost-WPIBC outperform the accuracy of DeepJIT by 16%, 15%, and 15%, respectively. Similarly, with the TV approach, JITBoost models surpass DeepJIT model by 10%. It is worth noting that both JITBoost models and DeepJIT show only small effects in performance based on the choice of data splitting approach. These results indicate that JITBoost models consistently outperform DeepJIT, regardless of the data splitting method employed.

C. RQ3: How does the combination of traditional JIT-SDP models and deep learning models affect the performance of the JITBoost algorithms?

This research question examines the impact of integrating DeepJIT predictions into JITBoost models to enhance their performance. The predictions generated by DeepJIT are combined with ML models for this purpose. However, it is worth noting that previous discussions have already addressed the influence of data-splitting approaches on JITBoost models. Hence, our objective is to measure the extent of improvement achieved by incorporating DeepJIT.

The overall performance of JITBoost models, with and without DeepJIT, is presented in Figure 6. We found that including DeepJIT as the seventh classifier into JITBoost does not affect JITBoost-BBC models using both CV and TV techniques. The JITBoost-IBC models improved by 1% when we include DeepJIT with CV, but the performance decreased by -1% using TV. Finally, The JITBoost-WPIBC is improved by 4% and 2% using both CV and TV, respectively, when the DeepJIT predictions are included. The p-value for all models is greater than 0.05, leading us to not reject the null hypothesis (h_0) and indicating no statistically significant differences in the results. Additionally, the effect size, as indicated by Cliff's δ, is observed to be small.

As discussed in Section III-1, the JITBoost-BBC algorithm generates all possible combinations of boolean functions leading to the best output, which does not effected when

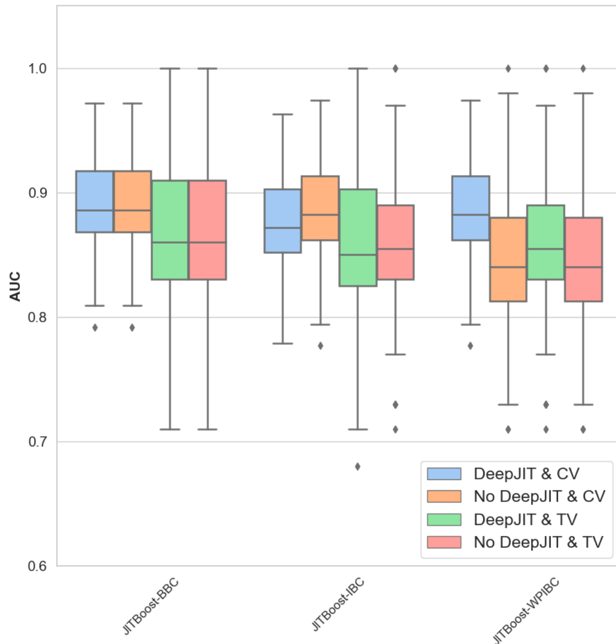

Fig. 6: Comparison of JITBoost models with DeepJIT to JITBoost models with and without DeepJIT.

incorporating the DeepJIT model compared to the JITBoost-IBC and JITBoost-WPIBC models. Also, the DeepJIT does not outperform the RF model in the preivous results. However, it is important to acknowledge that the BBC algorithm's major drawback is its high processing complexity. On the other hand, the IBC and WPIBC algorithms optimize their processing time by pruning possible cases and accelerating the combination of Boolean functions [11]. This difference in approaches explains the improvement observed when adding DeepJIT as a classifier to JITBoost-WPIBC models, because WPIBC drop out the similar vectors from the predictions.

TABLE VI: The effect size of improvement gain after combining all seven models.

Model	IM_{CV}	IM_{TV}	Cliff's δ	p-value
JITBoost-BBC	0%	0%	0.267	0.058
JITBoost-IBC	1%	-1%	0.266	0.060
JITBoost-WPIBC	4%	2%	0.262	0.063

Moreover, it is important to consider the computational resources required for training DL models. As illustrated in Table VII, this study uses different hardware configurations, including 2 CPUs and 3 GPUs, for tuning, training, and testing the DeepJIT model. The optimal hardware configuration in this study was identified as C_2 and G_2. These powerful hardware setups were specifically employed to accelerate the processing time of the DL model. Conversely, a simpler hardware configuration, such as C_1, was sufficient for the ML model.

TABLE VII: Hardware specifications utilized for deep learning model.

ID	Hardware	Specifications	Release Date
C_1	CPU Ryzen 9 5900X	12 cores, 3.7 GHz, 32GB RAM-DDR4	Nov-2020
C_2	CPU Intel Xeon	48 cores, 3.8 GHz, 64GB RAM-DDR4L	Jun-2019
G_1	GPU GTX 970	1664 CUDA cores, 4 GB GDDR5 RAM	Sep-2014
G_2	GPU RTX 8000	4608 CUDA cores, 48 GB GDDR6 RAM	Aug-2018
G_3	GPU A100 NVIDIA	6912 CUDA cores, 40 GB HBM2 RAM	May-2020

Finding RQ3: We found that JITBoost models outperform DeepJIT when compared to other ML models. Even when considering DeepJIT as a seventh classifier alongside the JITBoost models, the observed improvement is not statistically significant. It is important to note that training the DL model like DeepJIT requires substantial computing resources and time. While JITBoost models demonstrate better performance without such resource-intensive training requirements, the difference in performance between JITBoost and DeepJIT are not statistical significant.

VI. Threats to Validity

In this section, the threats to the validity of our results and recommendations is discussed.

Internal Validity: Internal validity threats refer to factors that could potentially influence our findings. One potential threat is the choice of algorithms. To address this, we employed robust algorithms that have a strong track record in various classification tasks and are widely used in research across different fields. Another concern relates to the datasets we used. Although we conducted experiments on 34 different Java Apache projects, incorporating additional datasets comprising different programming languages would enhance the generalizability of our results. Furthermore, comparing cross-validation with time validation might be impacted by the number of tests conducted. Time validation has certain limitations that prevent multiple tests similar to cross-validation. Additionally, the choice of the number of folds in cross-validation can also have an influence.

External Validity: External validity pertains to the extent to which our findings can be generalized. Our experiments encompassed 34 datasets from diverse software projects. However, it is important to note that we do not make claims regarding the generalizability of our results to all projects, especially industrial or proprietary systems to which we did not have access.

VII. Replication Package

All the data, scripts, and results discussed in this paper are available on Zenodo: **https://doi.org/10.5281/zenodo.8206280**

VIII. Conclusion

In this study, we examined the effectiveness of Boolean Combination Classifiers (BCC) for Just-In-Time Software Defects Prediction (JIT-SDP) models. Our experiments involved three BCC algorithms: Brute-force Boolean Combination

(BBC), Iterative Boolean Combination (IBC), and Weighted Pruning Iterative Boolean Combination (WPIBC), using 34 datasets. We compared their performance against state-of-the-art JIT-SDP models that utilize machine learning (ML) and deep learning (DL) approaches.

Our findings revealed that, in our specific case, DeepJIT was unable to outperform certain ML models, such as Random Forest (RF). However, when combining ML models within the JITBoost framework, the JITBoost algorithms outperformed all state-of-the-art JIT-SDP models employing ML and DL classifiers. Notably, including DL models alongside JITBoost algorithms enhanced the performance of the JITBoost-WPIBC algorithm. Furthermore, we observed that the choice of ML models significantly impacted data-splitting approaches, such as cross-validation and time-aware validation. In contrast, DL and JITBoost models exhibited minimal effects in this regard.

References

[1] M. Nayrolles and A. Hamou-Lhadj, "Clever: Combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR'18)*, 2018, pp. 153–164.

[2] R. Duan, H. Xu, Y. Fan, and M. Yan, "The impact of duplicate changes on just-in-time defect prediction," *IEEE Transactions on Reliability*, vol. 71, no. 3, pp. 1294–1308, 2022.

[3] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2013.

[4] C. Pornprasit and C. K. Tantithamthavorn, "Jitline: A simpler, better, faster, finer-grained just-in-time defect prediction," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, vol. 1, 2021, pp. 369–379.

[5] G. Catolino, D. Di Nucci, and F. Ferrucci, "Cross-project just-in-time bug prediction for mobile apps: An empirical assessment," in *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 2019, pp. 99–110.

[6] M. A. Shehab, A. Hamou-Lhadj, and L. Alawneh, "Clustercommit: A just-in-time defect prediction approach using clusters of projects," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022, pp. 333–337.

[7] T. Hoang, H. Khanh Dam, Y. Kamei, D. Lo, and N. Ubayashi, "Deepjit: An end-to-end deep learning framework for just-in-time defect prediction," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 34–45.

[8] T. Hoang, H. J. Kang, D. Lo, and J. Lawall, "Cc2vec: Distributed representations of code changes," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, vol. 1, 2020, pp. 518–529.

[9] Z. Zeng, Y. Zhang, H. Zhang, and L. Zhang, "Deep just-in-time defect prediction: How far are we?" in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 427–438. [Online]. Available: https://doi.org/10.1145/3460319.3464819

[10] W. Zheng, T. Shen, X. Chen, and P. Deng, "Interpretability application of the just-in-time software defect prediction model," *Journal of Systems and Software*, vol. 188, p. 111245, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121222000218

[11] W. Khreich, E. Granger, A. Miri, and R. Sabourin, "Boolean combination of classifiers in the roc space," in *2010 20th International Conference on Pattern Recognition*, 2010, pp. 4299–4303.

[12] W. Khreich, B. Khosravifar, A. Hamou-Lhadj, and C. Talhi, "An anomaly detection system based on variable n-gram features and one-class svm," *Information and Software Technology*, vol. 91, pp. 186–197, 2017.

[13] M. S. Islam, W. Khreich, and A. Hamou-Lhadj, "Anomaly detection techniques based on kappa-pruned ensembles," *IEEE Transactions on Reliability*, vol. 67, no. 1, pp. 212–229, 2018.

[14] W. Li, W. Zhang, X. Jia, and Z. Huang, "Effort-aware semi-supervised just-in-time defect prediction," *Information and Software Technology*, vol. 126, pp. 106 – 364, 2020.

[15] F. Lomio, L. Pascarella, F. Palomba, and V. Lenarduzzi, "Regularity or anomaly? on the use of anomaly detection for fine-grained just-in-time defect prediction," in *30th IEEE/ACM International Conference on Program Comprehension (ICPC 2022)*, vol. 1, 05 2022, pp. 1–10.

[16] L. Pascarella, F. Palomba, and A. Bacchelli, "Fine-grained just-in-time defect prediction," *Journal of Systems and Software*, vol. 150, pp. 22 – 36, 2019.

[17] W. Khreich, S. S. Murtaza, A. Hamou-Lhadj, and C. Talhi, "Combining heterogeneous anomaly detectors for improved software security," *Journal of Systems and Software (JSS)*, vol. 137, pp. 415–429, 2018.

[18] M. Barreno, A. Cardenas, and J. D. Tygar, "Optimal roc curve for a combination of classifiers," *Advances in Neural Information Processing Systems*, vol. 20, 2007.

[19] W. Khreich, E. Granger, A. Miri, and R. Sabourin, "Iterative boolean combination of classifiers in the roc space: An application to anomaly detection with hmms," *Pattern Recognition*, vol. 43, no. 8, pp. 2732–2752, 2010. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0031320310001263

[20] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction," *IEEE Transactions on Software Engineering*, vol. 44, no. 5, pp. 412–428, May 2018.

[21] Y. Fan, X. Xia, D. A. da Costa, D. Lo, A. E. Hassan, and S. Li, "The impact of mislabeled changes by szz on just-in-time defect prediction," *IEEE Transactions on Software Engineering*, vol. 47, no. 8, pp. 1559–1586, 2021.

[22] E. C. Neto, D. A. da Costa, and U. Kulesza, "The impact of refactoring changes on the szz algorithm: An empirical study," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 380–390.

[23] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proceedings of the 2005 International Workshop on Mining Software Repositories*, ser. MSR '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 1–5. [Online]. Available: https://doi.org/10.1145/1083142.1083147

[24] L. Yang and A. Shami, "On hyperparameter optimization of machine learning algorithms: Theory and practice," *Neurocomputing*, vol. 415, pp. 295–316, 2020.

[25] M. Feurer and F. Hutter, *Hyperparameter Optimization*. Cham: Springer International Publishing, 2019, pp. 3–33.

[26] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in *Proceedings of the 37th International Conference on Software Engineering Volume2*, ser. ICSE '15. IEEE Press, 2015, p. 99–108.

[27] C. M. Bishop, *Pattern recognition and machine learning*, ser. Information science and statistics. New York, NY: Springer, 2006.

[28] P. Bruce and A. Bruce, *Practical statistics for data scientists: 50 essential concepts*. O'Reilly Media, Inc., 2017.

[29] T. Fawcett, "An introduction to roc analysis," *Pattern Recognition Letters*, vol. 27, no. 8, pp. 861–874, 2006, rOC Analysis in Pattern Recognition. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016786550500303X

[30] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.

[31] A. Bucchianico, "Combinatorics, computer algebra and the wilcoxon-mann-whitney test," *Journal of Statistical Planning and Inference*, vol. 79, no. 2, pp. 349–364, 1999. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0378375898002614

[32] G. D. Ruxton, "The unequal variance t-test is an underused alternative to Student's t-test and the Mann–Whitney U test," *Behavioral Ecology*, vol. 17, no. 4, pp. 688–690, 05 2006. [Online]. Available: https://doi.org/10.1093/beheco/ark016

[33] J. Romano, J. Kromrey, J. Coraggio, J. Skowronek, and L. Devine, "Exploring methods for evaluating group differences on the nsse and other surveys: Are the t-test and cohensd indices the most appropriate choices," in *Annual meeting of the Southern Association for Institutional Research*, 2006.

[34] J. Cohen, "A power primer," *Psychological Bulletin*, vol. 112, pp. 155–159, 1992.