Machine Learning-based Test Case Prioritization using Hyperparameter Optimization

Md Asif Khan
Akramul Azim
Ramiro Liscano
Ontario Tech University
Oshawa, Ontario, Canada
mdasif.khan@ontariotechu.ca
akramul.azim@ontariotechu.ca
ramiro.liscano@ontariotechu.ca

Kevin Smith
Qasim Tauseef
Gkerta Seferi
IBM United Kingdom Limited
Portsmouth, Hampshire, United Kingdom
smithk6@uk.ibm.com
Qasim.Tauseef@ibm.com
Gkerta.Seferi@ibm.com

Yee-Kang Chang
IBM Canada Ltd.
Markham, Ontario, Canada
yeekangc@ca.ibm.com

Abstract

Continuous integration pipelines execute extensive automated test suites to validate new software builds. In this fast-paced development environment, delivering timely testing results to developers is critical to ensuring software quality. Test case prioritization (TCP) emerges as a pivotal solution, enabling the prioritization of fault-prone test cases for immediate attention. Recent advancements in machine learning have showcased promising results in TCP, offering the potential to revolutionize how we optimize testing workflows. Hyperparameter tuning plays a crucial role in enhancing the performance of ML models. However, there needs to be more work investigating the effects of hyperparameter tuning on TCP. Therefore, we explore how optimized hyperparameters influence the performance of various ML classifiers, focusing on the Average Percentage of Faults Detected (APFD) metric. Through empirical analysis of ten real-world, large-scale, diverse datasets, we conduct a grid search-based tuning with 885 hyperparameter combinations for four machine learning models. Our results provide model-specific insights and demonstrate an average 15% improvement in model performance with hyperparameter tuning compared to default settings. We further explain how hyperparameter tuning improves precision (max = 1), recall (max = 0.9633), F1-score (max = 0.9662), and influences APFD value (max = 0.9835), indicating a direct connection between tuning and prioritization performance. Hence, this study underscores the importance of hyperparameter tuning in optimizing failure prediction models and their direct impact on prioritization performance.

Keywords

hyperparameter optimization, test case prioritization, machine learning, continuous integration

AST '24, April 15–16, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0588-5/24/04…$15.00
https://doi.org/10.1145/3644032.3644467

ACM Reference Format:
Md Asif Khan, Akramul Azim, Ramiro Liscano, Kevin Smith, Qasim Tauseef, Gkerta Seferi, and Yee-Kang Chang. 2024. Machine Learning-based Test Case Prioritization using Hyperparameter Optimization . In *5th ACM/IEEE International Conference on Automation of Software Test (AST 2024) (AST '24), April 15–16, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3644032.3644467

1 Introduction

Continuous Integration (CI) is a widely used methodology that automates software development's compilation, building, and testing phases to minimize integration issues and reduce overall development time. Whenever developers integrate their code changes with the mainline codebase in CI, it triggers a CI cycle, and they receive regression test results after the compilation of each cycle. The compilation time of each cycle depends on the volume or complexity of tests, which can vary based on the project code base. If the volume or the complexity of tests is significant, it requires substantial computational resources and time to execute, resulting in delayed CI builds and longer feedback time for developers. Hence, the testing process directly affects the cost of software development, making it a crucial aspect of the development cycle. To this end, the test case prioritization (TCP) technique is implemented to execute fault-prone test suites earlier using heuristics or machine learning classification techniques.

Machine learning models significantly impact the quality of prioritization outcomes, particularly in the context of TCP. Nevertheless, these classification models include a layer of customization through hyperparameters. By adjusting these hyperparameters, the performance and characteristics of the classifiers they produce can be optimized. For instance, the number of decision trees comprising the forest can be altered when utilizing a random forest classifier. A similar requirement applies to k-nearest neighbors classifiers: the number of non-overlapping clusters must be configured. The optimization of these hyperparameters is facilitated by two standard methods: grid search and random search. While both methods involve an exhaustive exploration of hyperparameter combinations, grid search systematically tests every combination, making it more comprehensive than random search. This systematic approach is often preferred as it ensures a thorough evaluation of the hyperparameter space, leading to more precise hyperparameter settings and improved model performance. It is critical to acknowledge that in

the absence of suitable hyperparameter tuning, the performance of these machine learning models in effectively prioritizing test cases may be compromised, thereby hindering their ability to function optimally.

Despite the importance of these parameters in influencing the performance of classification methodologies, identifying the most effective configurations before training continues to present a difficult challenge. As a result, parameter values, by default, are frequently maintained. However, previous research has indicated that implementing this approach could potentially result in inadequate efficacy of failure prediction models. Many studies, including those by Jiang et al. [1] and Tosun et al. [2] on selecting and random forest, have emphasized the insufficient quality of the default parameter values utilized by classifiers such as naive Bayes and random forest. The significance of parameter settings is underscored by the fact that at least one parameter must be configured for most of the 30 most popular classification approaches, or 87 percent. This observation underscores the significance of careful parameter selection; however, it is not feasible to perform an exhaustive examination of each possibility within the parameter space of an individual classification algorithm. Kocaguneli et al. [3] provide a compelling illustration of this matter through their research, demonstrating the vast array of possibilities and establishing that training the k-nearest neighbors classifier necessitates traversing a minimum of 17,000 possible configurations.

The research conducted by Aydin et al. [4], Tantithamthavorn et al. [5, 6], and Osman et al. [7] has provided significant insights into the relationship between hyperparameters and the performance metrics of machine learning classifiers. Nevertheless, these optimizations' direct effects on test case prioritization outcomes were beyond the scope of their research. As a result, although these studies offer valuable insights into optimizing hyperparameters for classifiers, they must address the specific requirements and challenges of TCP to fill the existing gap in our comprehension of this crucial aspect of software testing. Our study aims to address this disparity by concentrating solely on the impacts of hyperparameter tuning on TCP, thereby providing a distinct and customized viewpoint on this matter.

We conducted a comprehensive empirical investigation utilizing a wide range of ten datasets obtained from proprietary and open-source domains, comprising a maximum of 4.5 million results from test suite executions. The present study used four distinct machine-learning models and examined the impact of twelve different hyperparameter settings, following thorough guidance from state-of-the-art recommendations. This comprehensive investigation yielded 885 distinct combinations of machine learning models. Our study has provided compelling evidence that hyperparameter tuning offers significant benefits, mainly when applied to extensive CI datasets. This is achieved through the systematic utilization of grid search-based hyperparameter evaluation. The results of our study demonstrate that the careful adjustment of hyperparameters significantly influences the process of test case prioritization, leading to a notable improvement in its effectiveness.

The major contributions of our findings are as follows :

(1) **Grid search-based Hyperparameter tuning on large CI dataset**: For four machine learning models, we conducted a grid search with 885 combinations of hyperparameters in ten diverse real-world datasets, comprising a maximum of 4.5 million test results, showcasing the thoroughness of our analysis and the depth of our exploration. This extensive dataset variation ensures that our findings are robust and applicable across large-scale datasets.
(2) **Improvement in failure prediction**: Through empirical analysis, we demonstrate that optimized hyperparameters lead to substantial enhancements in model performance, emphasizing the broad applicability of this technique. We show consistent improvements in precision (max = 1), recall (max = 0.9633), F1-score (max = 0.9662) and AUC (max = 0.9815) with hyperparameter tuning, with an average improvement of 15% across various performance metrics from non-tuned sttings. These improvements in traditional metrics underscore the enhanced failure prediction capability achieved through tuning.
(3) **Impact of Hyperparameter Tuning on APFD**: We explore how hyperparameter tuning influences the APFD metric, a crucial measure in test case prioritization. We achieved a maximum APFD score 0f 0.9835 using tuned Extreme Gradient boosting classifier. This findings reveal that increased in traditional metrics, attained through hyperparameter tuning, correspond with improved APFD scores, highlighting the direct connection between tuning and prioritization performance.

The remainder of the paper is organized as follows: Section 2 positions this paper concerning related work conducted in terms of hyperparameter tuning to improve TCP. Section 3 defines hyperparameter tuning for TCP as an optimization problem. This section also puts forward the research questions that we explored. Section 4 illustrates the architecture and approach of our case study. Section 5 presents our three research questions' results. Section 6 discusses the implications of our results in the context of improved TCP followed by the disclosure of the validity of our research in Section 7, and Section 8 concludes the paper.

2 Related Work

In a comprehensive study conducted by [8], ten algorithms were introduced for adoption in continuous integration (CI) practices. The research involved a thorough examination, comparing these algorithms using subjects from the Apache Commons project. However, for a fair comparison, default parameter values were employed, introducing potential invalidity threats. This is particularly notable because the performance of individual algorithms can significantly vary based on tuning. While the choice of strategy or category has a lesser impact, the text suggests that a tuning step using existing methods (such as grid or randomized search) may be necessary. This observation has also spurred further exploration into hyperparameter tuning.

As evident from recent scholarly works, hyperparameter tuning remains a critical area of investigation in machine learning classifiers for failure prediction. Yang et al. [9] emphasize the pivotal role of hyperparameter optimization in elevating the performance of fault detection models, particularly in dynamic software environments. Their research highlights that well-tuned hyperparameters

enhance accuracy and stability in failure prediction, emphasizing its practical significance.

Similarly, Zhou et al. [10] delved into the influence of hyperparameter tuning on the accuracy and efficiency of failure prediction within industrial software systems. Their findings demonstrate that appropriate hyperparameter configuration accurately identifies critical failures, underscoring its practical relevance in real-world applications. Similarly, Nguyen et al. [11] provide empirical substantiation that fine-tuning hyperparameters boosts the effectiveness of machine learning classifiers for software defect prediction, emphasizing its role in mitigating risks associated with software failures.

Recent investigations by Zhang et al. [12] explored the optimization of hyperparameters for deep learning models in the context of failure prediction, showcasing its potential to capture intricate patterns and anomalies in complex software systems. This underscores how hyperparameter tuning continues to be a focal point for advancing the accuracy and efficiency of failure prediction methods.

Tantithamthavorn et al. [6] extensively studied automated parameter optimization using various techniques like grid search, random search, genetic algorithms, and differential evolution across 26 categorization strategies. Their research focused on applying defect prediction algorithms to 18 datasets from NASA, proprietary, Apache, and Eclipse. As a result of automated parameter optimization, accuracy witnessed improvements of up to 40%, according to the results.

Osman et al. [7] explored grid search algorithms for optimizing KNN and SVM models. Notably, their results showed a substantial enhancement in prediction accuracy, with KNN achieving up to 20% improvement and SVM achieving up to 10% improvement. Tantithamthavorn et al. [5] further examined the Caret hyperparameter optimization method concerning 26 classification approaches. Their research revealed that Caret optimization could significantly elevate the accuracy of defect prediction models by up to 40%, with 35% of classifiers demonstrating increased stability. This study utilized 18 datasets from NASA, proprietary, Apache, and Eclipse. In the context of anomaly detection, Aydin et al. [4] employed a multi-objective artificial immune approach to improve SVM-RBF kernel parameters. Their method led to increases in performance accuracy.

Additionally, the works of Aydemir et al. [13], and Ramakrishnan et al. [14] have explored the impact of hyperparameter tuning on various machine learning models in the context of software defect prediction. Aydemir et al. conducted a comprehensive study on the effects of hyperparameter optimization techniques like grid search and random search on the performance of classifiers, providing insights into how tuning can improve classification results. Ramakrishnan et al. focused on ensemble-based models and demonstrated the effectiveness of hyperparameter tuning in enhancing the accuracy and robustness of defect prediction models.

Existing research in hyperparameter tuning has predominantly focused on defect prediction rather than failure prediction, overlooking specific challenges posed by the latter. Limited dataset diversity restricts the generalizability of findings, often drawn from specific domains. Addressing these gaps is crucial for a comprehensive understanding of hyperparameter tuning implications on machine learning classifier performance in failure prediction and TCP scenarios, offering valuable insights for research and practical applications.

3 Problem definition & research questions

3.1 TCP as an ML model optimization problem: Rothermel et al. [15] described the test case prioritization as:

- Given: A test suite (T), a collection of permutations (PT), and a function from PT to the real numbers (f) are given.
- Problem: Find $\tau \in PT$ such that $(\forall \tau^*)\ (\tau^* \in PT\ )\ (\tau^* \neq \tau)$ $[f(\tau) \geq f(\tau^*)]$.

In this definition, PT represents the set of potential prioritization (orders) of T, and f represents an objective function that, when applied to any such order, produces an award value for that order. We can measure the award value by calculating APFD for that prioritized list. The list with maximum APFD value is chosen as the most effective prioritized list.

When machine learning is utilized to improve TCP, the models, which have been trained using features extracted from the relevant dataset, generate failure probabilities for every test suite. Using these prediction probabilities, we sort the suites in descending order. This prioritization ensures that the test suite with the highest probability of failure occupies the top position. In contrast, the test suite with the lowest probability of failure is at the bottom of the list. We can obtain prioritized list for each type of ML models we are implementing and choose the one with maximize the APFD value.

As highlighted in the previous Section 2, the performance of these ML models depends on their hyperparameters, which, in turn, affect the prediction probability values generated for test suites. Therefore, we aim to optimize the value of APFD while considering practical constraints on time and, optionally, resource allocation for tuning the hyperparameters of the models. We assume that the maximum tuning time, denoted as $T_{\max}$, should not exceed the average feedback time required to provide feedback to developers in large-scale CI systems. Exceeding this time constraint would defeat the purpose of prioritization. While resource constraints might not be an issue for most companies due to the abundance of virtual machines, we acknowledge the existence of this issue. Thus, we formulate the problem of improving TCP by hyperparameter tuning as an optimization problem, as follows:

Objective: Maximize APFD for each $X_{i,j}$ so that $f(\tau_{i,j}) \geq f(\tau^*)$ for different hyperparameter combinations $X_{i,j}$.

Subject to:
$$\sum_{i=1}^{n}\sum_{j=1}^{m} T_{i,j} \cdot X_{i,j} \leq T_{\max} \quad \text{(Time Constraint)}$$
$$\sum_{i=1}^{n}\sum_{j=1}^{m} R_{i,j} \cdot X_{i,j} \leq R_{\max} \quad \text{(Resource Allocation, Optional)}$$

Decision Variables:

$X_{i,j}$: Hyperparameter settings through grid search for model i, combination j.

Parameters:

$T_{i,j}$: Time required to optimize hyperparameter $X_{i,j}$

$T_{\max}$: Maximum allowable time for hyperparameter tuning.

$R_{i,j}$: Resources required to optimize hyperparameter $X_{i,j}$

$R_{\max}$: Maximum allowable resource allocation (Optional).

This optimization problem uses grid search to find the best values for the hyperparameters ($X_{i,j}$), intending to increase the APFD value while staying within time and, if necessary, resource limits. This aligns with the objective of test case prioritization: to minimize feedback time for developers while effectively identifying and prioritizing fault-prone test cases at the top of the prioritized list. The grid search systematically explores several hyperparameter combinations to ensure fine-tuned selections for improved failure prediction.

In the context of hyperparameter optimization, Bischl et al. (2023) [16] and Agrawal et al. (2022) [17] have explored foundational aspects, algorithms, best practices, and open challenges. While grid search is a thorough method, alternative approaches exist within the AutoML domain. These alternatives explore the hyperparameter configuration space in a non-exhaustive and more efficient manner. However, both studies underscore that grid search remains a robust and widely used option.

The work by Bischl et al. emphasizes the foundations and challenges of hyperparameter optimization, providing valuable insights into the complexities of the search space [16]. Agrawal et al., on the other hand, advocate for more straightforward hyperparameter optimization approaches in software analytics, emphasizing the importance of efficiency [17]. While acknowledging the existence of alternatives, this study continues to assert the efficacy of grid search in our optimization problem because it is straightforward to implement and deterministic. It also provides a baseline performance for comparison with more sophisticated techniques.

3.2 Research Questions: In this study, we aim to understand the effects of optimization of hyperparameter tuning on failure prediction capabilities, the relationship between model performance metrics and APFD, and the impact of hyperparameter tuning on APFD values. To address this overarching goal, we explore the following research questions along with their respective hypotheses:

Hypothesis (H1): Hyperparameter tuning significantly impacts the failure prediction capabilities of machine learning models, leading to improvements in performance metrics such as precision, recall, F1-score, and AUC while prioritizing test suites.

(RQ1) How does hyperparameter tuning impact the failure prediction capabilities of ML models, as measured by precision, recall, F1-score, and AUC? Is there any trade-off between hyperparameter optimization cost and model performance in TCP?

Hypothesis (H2): There exists a measurable relationship between specific evaluation metrics used to assess machine learning model performance and the TCP evaluation metrics, particularly APFD. This relationship can be quantified to understand how model performance metrics relate to test case prioritization outcomes.

(RQ2 Is there a tangible relationship between specific evaluation metrics used to assess machine learning model performance and the TCP evaluation metrics, particularly APFD? If so, how can we quantify these relationships?

Hypothesis (H3): Tuning the hyperparameters of machine learning models has a discernible effect on APFD values, leading to changes in test case prioritization outcomes. The degree of change in APFD is associated with the extent of hyperparameter tuning.

(RQ3 What are the effects of tuning the hyperparameters of machine learning models on APFD values, and how do these changes influence test case prioritization? We aim to elucidate the dynamics between hyperparameter tuning and TCP outcomes.

Through a comprehensive exploration of these research questions and hypotheses, we aim to provide valuable insights into optimizing machine learning models for failure prediction and its subsequent impact on effective test case prioritization strategies.

4 Empirical evaluation

To investigate the impact of hyperparameter tuning on machine learning model performance and its subsequent influence on test case prioritization (TCP) metrics, we conducted a systematic study. The methodology consists of the following steps:

4.1 Data Collection: Yaraghietal et al. [18], Mendoza et al. [19] and Elsener et al. [20] thoroughly examined ML-based techniques using 25 datasets. Each dataset was carefully designed with a comprehensive set of 150 features. These are notably some of the most comprehensive and publicly available datasets, derived from analyzing over 20,000 open-source Java projects with five-star ratings. The inclusion criteria ensured that all subjects possessed a sufficient number of failed tests and a regression testing time exceeding 5 minutes, crucial factors for applying and evaluating TCP techniques. The datasets also represent source code size, ranging from 61k to 4.56M lines of code, with a median of 229k. Additionally, they cover the number of tests in each continuous integration (CI) build, ranging from 33 to 4368 with a median of 117, and the number of failed builds (6 to 343 with a median of 83).

We further refined the dataset as smaller datasets may result in unstable results, while large-scale CI systems likely contain more failed tests than passed tests. Hence, the datasets were chosen with running period over 200 days to mitigate a poor training phase. Following these selection criteria, we intended to secure a collection of datasets, as illustrated in Table 1, that embodies diversity, public accessibility, and authenticity in representing real-world software testing scenarios. We also added the Open Liberty dataset, which was provided by IBM.

4.2 Feature Extraction: To comprehensively capture the characteristics of the software projects, we collected features from two distinct sources: Continuous Integration test suite execution results and the Version Control System (GitHub). We can extract all the features for Open Liberty where test suite results are reported, but for the rest of the datasets, we do not have access to information regarding #Test Suites, #Files changed, #Lines Inserted and Deleted. Hence, we created synthetic data points similar to Mamata et al. [21].

4.2.1 CI Test Execution Features From the CI test execution results, we extracted a range of features to capture the testing history and

Table 1: Statistics of Study datasets

Project	Time Period (days)	CI Cycles	#Test Suites	#Test Suites Executions	#Test Suites Failures	Failure Rate
Open liberty	1168	6124	1158	4571.5K	17583	0.38%
IVU Cpp	267	3996	1240	3608.4K	25,973	0.72%
IVU_Java_2	699	3209	1278	3526.3K	7603	0.22%
buck	307	846	864	586.1K	1,511	0.26%
jcabi-github	872	809	201	398.1K	740	0.19%
cloudify	909	4973	116	283.6K	602	0.21%
sling	213	1403	304	268.1K	1,158	0.43%
okhttp	1423	3412	266	236.5K	939	0.40%
IVU_Java_1	313	943	279	178.7K	14,568	8.15%
Achilles	1114	642	627	139.9K	162	0.12%
DSpace	1043	1929	83	122.1K	1,697	1.39%

patterns. Here, a single test is referred to as a **Test Suite**. The features include the **Test Identifier** (a unique test suite identifier), **Execution Count** (the number of times a test suite has been executed), **Test Total** (the total number of test suites in the dataset), **Failure Count** (the number of times a test suite has failed), **Failure Rate** (the ratio of failed executions to total executions), **Last Failure Age** (the amount of CI builds since the occurrence of the last failure), **Last Failure** (the outcome of the most recent test suite execution, represented as a Boolean), and **Last Transition** (the amount of CI builds since the occurrence of the last transition between passing and failing).

4.2.2 Version Control System (GitHub) Features From the GitHub repository, we gathered features that reflect the software's development and code changes. These include the **Number of Files Changed** (the count of files modified in a commit), **Lines Inserted** (the total number of lines inserted in a commit), and **Lines Deleted** (the total number of lines deleted in a commit).

4.3 Hyperparameter Tuning:

4.3.1 ***Model Selection*** The study primarily investigated tree-based machine learning models, namely Extreme Gradient Boosting (XGBoost), Gradient Boosting (GBoost), Random Forest (RF), and Decision Trees (DT), as they exhibited exceptional performance in the task of TCP. This is consistent with findings from prior research, including that of Menzies et al. [22], which emphasized the adaptability and efficacy of tree-based models in capturing complex relationships in software testing data. The inherent interpretability, capacity to handle both categorical and numerical features, and ability to identify feature importance were some driving forces behind our choice [23]. The study's objective was to utilize these models to improve TCP strategies, thereby facilitating more simplified and effective software testing procedures.

Furthermore, Bertolino et al. [8] demonstrated that Multiple Additive Regression Trees (MART), gradient-boosted regression Trees, emerged as the best machine learning ranking model in the TCP context. In contrast, Yaraghi et al. [18] found that the RF ranking model outperformed others, including MART, a deviation from Bertolino et al.'s results. Consequently, they opted to use RF for all experiments throughout the study. It is essential to note that the primary focus of this study is the evaluation of these models in the context of enhancing Test Case Prioritization.

4.3.2 ***Hyperparameter selection and values*** Due to the impracticality of completely testing all feasible parameter combinations within the parameter spaces, we referred to state-of-the-art findings and domain-specific knowledge to identify relevant hyperparameters for each machine learning model. We systematically varied these hyperparameters to fine-tune the models. We use a time budget threshold of 24 hours in accordance with Kuhn's concept [24]. Table 2 shows the hyperparameter settings, including the default settings shown in bold.

Additionally, it's noteworthy that our hyperparameter selection process involved a more extensive approach. While Bertolino et al. [8] opted for default hyperparameter values, In contrast, Yaraghi and Mendoza et al. [18, 19] focused solely on investigating RF hyperparameters, specifically rtype, srate, bag, frate, tree, leaf, and shrinkage, with a strength value of 3. This expanded scope allowed us to consider a broader array of hyperparameters, contributing to a more comprehensive and nuanced optimization of the machine learning models in our study.

4.3.3 ***Baseline Model Training*** To investigate the impact of hyperparameter tuning on machine learning model performance, we initiated the study with the training the above mentioned models in their default configurations. By doing so, we could establish a benchmark for model effectiveness before any hyperparameter adjustments were made. Key performance metrics, including precision, recall, F1-score, and AUC, were recorded for each model based on their predictions on the test dataset.

This baseline training phase provided essential insights into the starting point of model performance, serving as a reference for the subsequent evaluations after hyperparameter tuning. It allowed us to quantify the initial capabilities of the models in their default forms before investigating the impact of hyperparameter adjustments.

4.3.4 ***Hyperparameter tuning*** A comprehensive analysis was conducted on each possible configuration of hyperparameters using Grid search method. To provide an example, suppose a model comprises two parameters, each of which has five potential values. We conducted an exhaustive investigation of every 25 possible permutations of hyperparameter configurations individually for 10 datasets, adhering to budget constraints until we exhausted the list. The classifier evaluation metrics were calculated for each of the hyperparameter configurations through the process of training a classifier on a specific subset of the training dataset. The performance of the model was evaluated through an examination of the data rows that were not included in the particular subset utilized during classifier training.

4.4 TCP using hyperparameter optimized models: In the next step, the optimized models are used to get a prediction probability value for each test suites. Once the prediction probabilities are obtained, the test suites are sorted in descending order based on these probabilities. This sorting places the test suites with the

Table 2: Machine Learning Model HyperParameters

Model	Parameters
RF	n_estimators: [1, 5, 10, 15, 30, 50, 100, 500, 1000], 'criterion': ["gini", 'entropy'], 'max_features': ['auto', 'sqrt', 'log2']
DT	max_depth: [2, 4, 5, 6, 7, 8, 10, 12, 25, 50, 100, 500], 'criterion': ['gini', 'entropy'], 'splitter': ['best', 'random'], 'max_features': ['auto', 'sqrt', 'log2']
XGBoost	'subsample': [0.6, 0.8, 1], 'eta': [0.1, 0.5, 0.3], 'max_depth': [3, 4, 5, 10, 50, 100, 500]
GBoost	"learning_rate": [0.1], n_estimators: [100], "max_depth": [4, 6, 8, 10, 25, 50, 100, 500]

highest probability of failure at the top of the list, while those with lower probabilities follow. The rationale behind this approach is to prioritize the execution of test suites that are deemed more likely to uncover faults or failures in the software.

Subsequently, the test suites are executed in the determined order, starting from the top of the list. This sequential execution is designed to expedite the detection of potential issues in the software. Following the execution of the test suites, the required metrics and APFD is calculated for evaluation purposes as shown in Algorithm 12.

4.5 Evaluation: We evaluated the model's performance using a comprehensive set of metrics for each hyperparameter configuration and dataset. Specifically, we recorded precision, recall, F1-score, and AUC to gauge the classifier's effectiveness in predicting failures. Additionally, using the prediction probabilities produced by the respective machine learning model, we calculated the APFD value for each prioritized test suite. It is crucial to note that precision, recall, and F1-score are threshold-dependent evaluation metrics, while AUC is threshold-independent. This inclusive approach allowed us to thoroughly and unbiasedly assess our models' efficacy.

Let T be a test suite that consists of n number of test cases, and Rothermel et al. [15] calculated the APFD of a prioritized test suite T as follows:

$$APFD = 1 - \frac{TF_1 + TF_2 + ...TF_n}{nm} + \frac{1}{2n} \quad (1)$$

where, m = number of failed test cases, n = number of test cases, and TF_i = Rank of the first test case that indicates the i^{th} fault in test suite T'. We can also use the number of test suite failures to calculate APFD when the number of detected faults in a test suite is unavailable. Elbaum et al. [25] plots the executed test suites rate and detected faults rate in the x and y-axis, respectively, to calculate APFD. The minimum value of APFD is 0, while the maximum value is 1. Higher APFD values exhibit that the faults are identified faster using fewer test suites.

We systematically recorded the maximum and minimum values for each metric across different configurations to identify the hyperparameter combinations that optimize the objective function. This approach enabled us to focus on hyperparameter combinations that maximize APFD while also pinpointing combinations that may not warrant further consideration, thereby conserving valuable research time. Algorithm **1** represents the working algorithm of collecting evaluation metrics from tuned model.

We conducted a Pearson correlation analysis across the diverse hyperparameter configurations to uncover associations between model performance metrics (precision, recall, F1-score, and AUC) and APFD values. Pearson correlation analysis measures the strength and direction of a linear relationship between two variables: model performance metrics and APFD. This analysis is essential as it allows us to quantitatively assess how changes in model performance metrics correlate with differences in test case prioritization outcomes. Identifying such correlations provides valuable insights into which performance metrics indicate improved prioritization effectiveness.

The Pearson correlation coefficient (r) measures the linear relationship between two variables. It ranges from -1 to 1, where, $r = 1$ implies a perfect positive linear relationship, $r = -1$ implies a perfect negative linear relationship, and $r = 0$ implies no linear relationship.

To further, examine the influence of hyperparameter tuning on APFD values, we compared the APFD results obtained from the baseline (untuned) models to those achieved after hyperparameter tuning. This comparative analysis allowed us to assess how hyperparameter tuning affects test case prioritization outcomes independently of changes in model performance metrics.

Algorithm 1: Evaluation of hyperparameter-tuned ML models in TCP

Data: List of test suites T, Test suite execution features F_t, Code change features F_c, Feature-engineered features F_{fe}, Set of models $M = \{DT, RF, XGBoost, GBoost\}$, Set of hyperparameter combinations $\mathcal{X}$ through Grid Search, Threshold budget j

Result: Min and max performance metrics, APFD values

foreach *model* M_i *in* M **do**
  foreach *hyperparameter combination* $X_{M_i} \in \mathcal{X}$ **do**
    Create an empty list for results;
    Create feature vectors $V(T_i)$ for each test suite in T;
    Split the dataset into D_{train} and D_{test};
    Fit model M_i with hyperparameters X_{M_i} to D_{train};
    Make predictions on D_{test};
    Calculate performance metrics (precision, recall, F1-score, AUC);
    Sort test suites based on predictions;
    Calculate APFD for prioritized test suites;
    Store min and max values of performance metrics;
return *Min and max performance metrics, APFD values for analysis*;

In summary, this methodology provided a systematic approach to explore the impact of hyperparameter tuning on machine learning model performance and its implications for test case prioritization.

It allowed us to address the research questions and test the hypotheses, ultimately providing valuable insights into optimizing software testing strategies.

5 Experimental results and analysis

5.1 Hyperparameter Tuning Impact on Performance: In this experiment, we investigate the impact of hyperparameter tuning on the failure prediction capabilities of machine learning models, as measured by precision, recall, F1-score, and AUC curve. Our findings provide strong evidence that hyperparameter tuning plays a significant role in improving the failure prediction performance of machine learning models. Across all datasets, we observed consistent trends in the performance metrics when comparing tuned models to their default counterparts. We have recorded the maximum value obtained through all the possible combinations of hypeprparamters in Table 3. From this Table we can see that, for precision and recall metrics, the tuned models consistently outperformed the default models. The precision of the tuned models exceeded that of the default models across all datasets. Similarly, the recall achieved by the tuned models was consistently higher than that of the default models. This indicates that hyperparameter tuning allows the models to achieve better balance between precision and recall, resulting in improved prediction accuracy and fewer false negatives.

The F1-score, which considers both precision and recall, also demonstrated notable improvement with hyperparameter tuning. Across datasets, the tuned models consistently achieved an 80% improvement in F1-score compared to the default models. This indicates that the tuned models strike a better balance between precision and recall, resulting in enhanced overall performance.

Interestingly, when considering the AUC metric, a different pattern emerged. The default models consistently achieved higher AUC scores compared to their tuned counterparts. This suggests that the default models excel in distinguishing between positive and negative instances, particularly in the context of the receiver operating characteristic (ROC) curve. However, it's important to note that a higher AUC score doesn't necessarily imply a better overall predictive performance, as it depends on the specific requirements of the task at hand.

Upon closer examination of the models in Table 3, we observed interesting trends regarding which models performed best in terms of specific metrics. The tuned DT consistently achieved the highest precision among all models, indicating its suitability for applications where minimizing false positives is crucial. On the other hand, RF consistently produced the highest performance in terms of recall and F1-score. For AUC, default RF consistently outperformed other models, indicating its strength in ranking instances.

5.2 Correlation among evaluation metrics and APFD: The initial observations from the Table 4, which presents the Pearson Correlation between various metrics and the Average Percentage of Faults Detected (APFD) for all machine learning models combined, indicate a consistent pattern across the metrics.

Firstly, all metrics—Precision, Recall, F1 score, and AUC—show positive correlations with APFD, suggesting that an increase in any of these metrics tends to correspond with an increase in fault detection performance. This aligns with the intuitive expectation that higher values of these evaluation metrics are indicative of better fault detection capabilities.

Table 3: ML Classifier Performance (max values) Before and After Hyperparameter Tuning

Datasets	Metric	Default	Tuned	Change (%)
Open liberty	Precision	0.9392	1.0000	+6.49%
	Recall	0.5898	0.6160	+4.46%
	F1-score	0.7154	0.7225	+0.98%
	AUC	0.7948	0.6254	-21.31%
IVU Cpp	Precision	0.9539	1.0000	+4.88%
	Recall	0.9183	0.9273	+0.98%
	F1-score	0.9357	0.9407	+0.53%
	AUC	0.9590	0.9441	-1.55%
IVU_Java_2	Precision	0.9038	0.9740	+7.78%
	Recall	0.8731	0.9001	+3.09%
	F1-score	0.8870	0.9004	+1.51%
	AUC	0.9364	0.7877	-15.84%
buck	Precision	0.9760	1.0000	+2.46%
	Recall	0.9500	0.9617	+1.23%
	F1-score	0.9628	0.9679	+0.53%
	AUC	0.9750	0.9741	-0.92%
jcabi-github	Precision	0.7384	1.0000	+35.47%
	Recall	0.4291	0.5304	+23.55%
	F1-score	0.5427	0.5316	-2.05%
	AUC	0.7144	0.6215	-13.01%
cloudify	Precision	0.8172	1.0000	+22.37%
	Recall	0.6471	0.8403	+29.84%
	F1-score	0.7170	0.7356	+2.60%
	AUC	0.8234	0.8380	+1.77%
sling	Precision	0.7131	0.7763	+8.86%
	Recall	0.5944	0.6594	+10.93%
	F1-score	0.6020	0.6080	+1.00%
	AUC	0.7962	0.7595	-4.59%
okhttp	Precision	0.8571	0.9091	+6.07%
	Recall	0.2332	0.3485	+49.45%
	F1-score	0.3508	0.3752	+6.96%
	AUC	0.6164	0.6017	-2.37%
IVU_Java_1	Precision	0.9194	0.9683	+5.32%
	Recall	0.7980	0.8287	+3.85%
	F1-score	0.8544	0.8755	+2.47%
	AUC	0.8959	0.8468	-5.47%
DSpace	Precision	0.9921	1.0000	+0.81%
	Recall	0.9528	0.9513	-0.16%
	F1-score	0.9685	0.9662	-0.24%
	AUC	0.9763	0.9666	-0.97%

Table 4: Correlation between Metrics and APFD (mean) for ML Models Combined

Metric	Correlation	Correlation Type	Strength
Precision	0.4508	Positive	Moderate
Recall	0.3872	Positive	Moderate
F1 score	0.4866	Positive	Moderate
AUC	0.4509	Positive	Moderate

Secondly, the correlations for all metrics are categorized as "Moderate," with correlation coefficients ranging from 0.3872 to 0.4866. This implies that there is a moderate linear relationship between the metrics and APFD. In practical terms, this suggests that while these metrics are positively associated with fault detection performance, they may not be strong enough indicators to predict APFD with high precision.

These initial findings suggest that, for the given set of machine learning models and datasets, there exists a moderate positive relationship between the evaluated metrics and APFD. However, it is essential to delve deeper into the dataset-specific and model-specific correlations, as presented in subsequent tables, to understand how these relationships vary in different contexts.

Table 5: Correlation for Different Metrics and ML Models

ML Model	Metric	Correlation	Correlation	Strength
DT	Precision	0.2724	Positive	Weak
	Recall	0.4613	Positive	Moderate
	F1 Score	0.4807	Positive	Moderate
	AUC	0.4640	Positive	Moderate
RF	Precision	0.4893	Positive	Moderate
	Recall	0.4022	Positive	Moderate
	F1 Score	0.4477	Positive	Moderate
	AUC	0.4087	Positive	Moderate
GBoosting	Precision	0.6396	Positive	Strong
	Recall	0.6065	Positive	Moderate
	F1 Score	0.6695	Positive	Strong
	AUC	0.7170	Positive	Strong
XGB	Precision	0.2361	Positive	Weak
	Recall	0.1579	Positive	Very Weak
	F1 Score	0.1855	Positive	Very Weak
	AUC	0.1640	Positive	Very Weak

5.2.1 ***Individual model*** Table 5 summarizes the Pearson correlations for individual machine learning models, including DecisionTree (DT), RandomForest (RF), XGBoost (XGBoost), and GradientBoosting (GBoost), across various metrics and their correlation with the APFD (mean) metric.

One of the most noteworthy findings is the substantial positive correlation observed between the GradientBoostingClassifier and both Precision (0.6396) and F1 Score (0.6695). These strong correlations signify that the predictions generated by the GradientBoostingClassifier are closely aligned with the APFD metric. This suggests that this model is exceptionally adept at identifying faults accurately and achieving a balance between precision and recall, a crucial consideration in fault detection. Therefore, it emerges as a robust choice for tasks where precise fault detection is paramount.

Conversely, the XGBClassifier displayed comparatively weaker correlations across all metrics, with the lowest correlation recorded in AUC (0.1640). This implies that the predictions made by the XGBClassifier have a tenuous connection to APFD, indicating room for improvement in its fault detection capabilities. While this model may still hold promise for specific applications, its performance in fault detection tasks might benefit from further refinement or customization.

5.2.2 ***Individual Dataset*** The correlation analysis in Table 6 provides crucial insights into the relationships between various evaluation metrics and the Average Percentage of Faults Detected (APFD) for different datasets. These findings have significant implications for understanding the performance of fault detection algorithms across diverse software datasets.

Firstly, for the "buck.csv" dataset, Precision exhibits a very weak positive correlation with APFD (0.0076), indicating that it has minimal influence on fault detection in this context. On the other hand, Recall demonstrates a strong positive correlation (0.6114) with APFD, implying that this metric is a robust indicator of fault detection performance for this dataset. F1 score and AUC also show noteworthy correlations, both classified as moderate (0.5728 and 0.6114, respectively), suggesting their relevance in assessing fault detection effectiveness.

The "cloudify.csv" dataset highlights strong correlations across all metrics, emphasizing the robustness of these evaluation measures for fault detection. Precision, Recall, F1 score, and AUC all exhibit strong positive correlations, making them valuable tools for assessing fault detection in this particular dataset.

Conversely, the "Open-Liberty" dataset presents a mixed picture. While Precision and Recall show strong positive and weak correlations, respectively, with APFD, the F1 score and AUC exhibit moderate and very weak correlations, respectively. This dataset poses a unique challenge, where different metrics provide varying levels of insight into fault detection performance.

To gain a more comprehensive understanding of these relationships, Table 7 quantifies the distribution of correlation strengths across different metrics. Notably, 60% of the metrics show moderate correlations, while 30% exhibit strong correlations. Very weak correlations are observed in 30% of cases, primarily for Recall and AUC, indicating that these metrics may not be as effective in some scenarios. Weak correlations are relatively less common, with only 10% for Recall and AUC. These insights underscore the importance of carefully selecting evaluation metrics based on dataset characteristics and research goals, as their effectiveness can vary significantly.

5.2.3 ***Change in APFD value after hyperparameter tuning*** To investigate the effects of tuning the hyperparameters of machine learning models on APFD values we aim to elucidate the dynamics between hyperparameter tuning and TCP outcomes. for each hyperparameter combinations of ML model we plot the APFD value in Figure 1. We can see that the DecisionTreeClassifier demonstrated a mixed performance, with a minimum APFD (mean) of 0.5069, showcasing its potential for less effective test prioritization.

Table 6: Pearson Correlation between Different Metrics and APFD (mean) for Various Datasets

Metric	buck.csv	cloudify.csv	cpp.csv	dspace.csv	Open-Liberty	java1.csv	java2.csv	jcabi.csv	okhttp.csv
Precision	Very Weak	Strong	Strong	Moderate	Weak	Very Strong	Moderate	Very Weak	Moderate
Recall	Strong	Moderate	Strong	Moderate	Weak	Very Strong	Moderate	Very Weak	Very Weak
F1 Score	Moderate	Strong	Strong	Moderate	Moderate	Very Strong	Strong	Very Weak	Moderate
AUC	Strong	Strong	Strong	Moderate	Very Weak	Very Strong	Moderate	Very Weak	Very Weak

Table 7: Correlation Strength (%) for Different Metrics

Metric	Very Weak	Weak	Moderate	Strong	Very Strong
Precision	10%	0%	60%	30%	0%
Recall	30%	10%	30%	20%	10%
F1 score	10%	0%	50%	30%	10%
AUC	30%	0%	30%	30%	10%

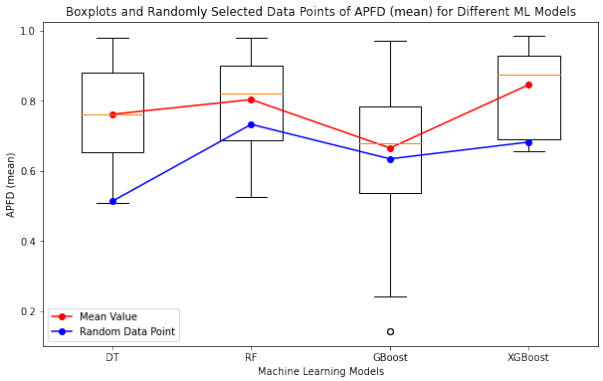

Figure 1: APFD values for different hyperparameter combinations of ML models

However, it also achieved a notable maximum APFD of 0.9791, suggesting promise in effectively identifying fault-prone test cases. The median APFD, standing at 0.7595, indicated a moderate overall performance.

In contrast, the RandomForestClassifier model displayed a more consistent TCP performance. It yielded a minimum APFD (mean) of 0.5242 and a maximum of 0.9804, with a robust median APFD of 0.8211, signifying strong and reliable test case prioritization. The GradientBoostingClassifier exhibited a broader range of APFD values, with a minimum of 0.1425 and a maximum of 0.9708. The median APFD, calculated at 0.6774, indicated a moderate overall performance.

Finally, the XGBClassifier model demonstrated a highly consistent and robust TCP strategy. It produced a minimum APFD (mean) of 0.6567 and a maximum of 0.9835, with a commanding median APFD of 0.8746, showcasing strong and reliable test case prioritization.

The comparison of the mean APFD values between randomly selected hyperparameter models (blue line) and hyperparameter-tuned models (red line) for Decision Trees (DT), Random Forest (RF), Gradient Boosting (GBoost), and XGBoost reveals important insights. In this analysis, it's evident that hyperparameter tuning significantly improves the performance of machine learning models. The red lines, representing tuned models, consistently exhibit higher mean APFD values compared to the blue lines, which represent random hyperparameter assignments. This suggests that a systematic approach like grid search-based hyperparameter tuning is essential for achieving better results in optimizing these machine learning models. Therefore, it's clear that hyperparameter optimization is a critical step in model development, as it can lead to substantial performance improvements and should not be left to random assignment .

6 Discussion

6.1 RQ1: To asnwer to RQ1 we find that hyperparameter tuning significantly enhances the predictive abilities of machine learning models. It leads to improved precision and recall metrics compared to default models, resulting in better accuracy and fewer false negatives. Tuned models also outperformed default models in terms of F1-score, demonstrating an 80% enhancement in F1-score. However, default models consistently outperformed tuned models in terms of AUC scores, suggesting they excel in distinguishing between positive and negative instances. The effectiveness of a model depends on the specific requirements of the task.

6.1.1 ***Hyperparameter optimization cost*** Analyzing Table 8 provides valuable insights into the costs associated with hyperparameter tuning across various machine learning models and configurations. For Decision Trees, the tuning time is relatively low, ranging from a few seconds to around 18 seconds. The time increases with higher values of the `max_depth` parameter, regardless of the chosen criteria (`gini` or `entropy`). In the case of Random Forest, the tuning time significantly increases as the number of estimators (`n_estimators`) grows, spanning from a few seconds to several hours, with longer times observed for higher values of `n_estimators`. Gradient Boost tuning times vary based on the `max_depth` and other parameters, showing a broader range compared to Decision Trees. Elevated values of `max_depth`, along with increased `n_estimators` and learning rate (`LR`), generally result in extended tuning times. Finally, for XGBoost, hyperparameter tuning generally entails higher times, especially for larger values of `max_depth`, with the tuning time substantially increasing and reaching its peak for `max_depth=5,000`.

We can conclude that ensemble methods like RF and XGBoost require more computational resources for hyperparameter tuning than individual Decision Tree models. The time complexity is influenced by hyperparameters like max_depth, n_estimators, and LR.

Table 8: Hyperparameter tuning cost Comparison for Different Models

Model	Hyperparameter Combination	Time (s)
DT	criterion=gini, max_depth=2	3.78
DT	criterion=entropy, max_depth=2	4.04
DT	criterion=gini, max_depth=4	6.21
DT	criterion=entropy, max_depth=4	6.00
DT	criterion=gini, max_depth=50	18.9
DT	criterion=entropy, max_depth=50	18.25
DT	criterion=gini, max_depth=100	18.56
DT	criterion=entropy, max_depth=100	18.12
RF	n_estimators=1	4.87
RF	n_estimators=5	20.95
RF	n_estimators=500	2292
RF	n_estimators=1000	4578.70
RF	n_estimators=5000	22929
GBoost	max_depth=4	266.65
GBoost	max_depth=6	272.29
GBoost	max_depth=10, n_estimators=500, LR=0.1	311.98
GBoost	LR=1.0, max_depth=10, n_estimators=100	438
GBoost	LR=1.0, max_depth=10, n_estimators=500	417
GBoost	max_depth=500	358.42
XGBoost	max_depth=3	45.46
XGBoost	max_depth=5	77.43
XGBoost	max_depth=10	172.34
XGBoost	max_depth=50	854.71
XGBoost	max_depth=100	1107
XGBoost	max_depth=500	1113
XGBoost	max_depth=1,000	1084
XGBoost	max_depth=5,000	1092.6

Larger hyperparameter values lead to longer tuning times, emphasizing the trade-off between model complexity and computational cost. The study emphasizes the need for efficient tuning strategies to balance optimal model performance and computational resource management.

6.2 RQ2: RQ2 investigates the associations between different evaluation metrics and the APFD metric in ML models across various fault prediction contexts. Across all models and datasets, positive correlations were observed between Precision, Recall, F1 Score, AUC, and APFD. However, these correlations were moderate, suggesting they may not precisely predict APFD. GBoost exhibited strong correlations across all metrics, while XGBClassifier showed weaker correlations, especially in AUC. Dataset-specific analysis revealed varying correlations, with some datasets showing solid and consistent associations and others exhibiting mixed correlations. Overall, moderate and strong correlations were prevalent, emphasizing their reliability, but very weak correlations were observed in 30% of cases, particularly for Recall and AUC. This underscores the importance of selecting metrics tailored to the dataset and research goals, providing valuable guidance for researchers and practitioners in fault prediction tasks.

6.3 RQ3: The study investigates the impact of hyperparameter tuning on the APFD values of machine learning models. The DT classifier exhibited a performance spectrum marked by fluctuations in its diverse configurations, indicating a sensitivity to hyperparameter settings. In stark contrast, the RF classifier showcased a more steadfast and consistent performance in TCP, underlining its resilience to variations in hyperparameter values. Moving beyond, the GBoost classifier presented a broader range of APFD values, suggesting a more intricate sensitivity to hyperparameter configurations. In the case of the XGBoost classifier, notable consistency and robustness in its TCP strategy were observed across various hyperparameter settings. Collectively, these insightful findings contribute to a deeper comprehension of the intricate interplay between hyperparameter tuning, the selection of machine learning models, and the dynamics of test case prioritization.

6.3.1 ***Navigating Potential Overfitting and Underfitting*** The variations in performance observed across different hyperparameter combinations in Table 3 indicate potential overfitting or underfitting, underscoring the importance of a nuanced approach to hyperparameter tuning. For instance, the DT with specific hyperparameters achieving a perfect precision score may suggest overfitting, as the model could be overly complex, capturing noise as meaningful patterns. Conversely, the DT model with a different configuration displaying zero precision may indicate underfitting, suggesting the model is too simplistic to discern true positives. A similar balance is seen in the recall scores, where the GBoost excels with specific hyperparameters, possibly avoiding overfitting while effectively identifying positive instances. The highest F1 Score in the RF indicates a balanced precision-recall trade-off, while a zero F1 Score in another DT configuration suggests underfitting. The APFD values highlight potential overfitting or insensitivity to faults in specific GBoost configurations, contrasting with the effective fault detection capabilities of the XGBClassifier. Similarly, the Area under the AUC in GBoost models reveals potential overfitting in one case and robust predictive capabilities in another. Striking the right balance in hyperparameter tuning is crucial to optimize model complexity, preventing overfitting or underfitting and ensuring generalization to new data.

7 Threats to Validity

7.1 Construct Validity: We used data from various sources with different methods and class breakdowns, which could affect the findings. However, testing datasets with the same evaluation metrics showed that changing the number or type of predictors didn't significantly alter the results. Using datasets with different metrics and feature breakdowns made the results more general, demonstrating they are not tied to just one type of evaluation metric. The study also limited the number of settings to try, which could change the results. Despite choosing different limits, the results remained consistent across all tests, indicating that changing the limit wouldn't significantly alter the main findings. We carefully planned the study design, considering differences in evaluation metrics and limits, to make the results more reliable and applicable to different situations.

7.2 Internal Validity: This paper explores the impact of parameter optimization on specific software systems, highlighting its

significance in specific contexts. The study's limitations include not extending to all software systems, including open-source and commercial ones. The evaluation of automated parameter optimization techniques is based on performance enhancements, but the selection of these techniques should consider factors such as computational costs, customization potential, and sensitivity of optimization technique parameters. A comprehensive assessment of these multi-dimensional aspects is crucial for informed decision-making regarding the adoption of automated parameter optimization techniques in real-world scenarios. Further replication studies could provide further clarity and confirm the applicability of the findings to a broader range of software scenarios.

7.3 External Validity: The discussion of the experimental outcomes involves the assessment of ten distinct datasets, predominantly composed in Java. Our experimental procedure takes into account designated attributes extracted from Continuous Integration (CI) and GitHub repositories. It's worth noting that the outcomes might differ when different sets of attributes are chosen. Nevertheless, we are confident that our study can be replicated using identical model configurations and the dataset provided in the repository.

8 Conclusion

The study reveals that hyperparameter tuning and performance metrics are crucial in failure prediction and test case prioritization. By adjusting hyperparameters, models can improve key performance metrics, such as the F1 Score, which leads to more accurate fault detection. The study also found that AUC values, which indicate the model's ability to discriminate between faulty and non-faulty cases, significantly influence prioritization effectiveness. The findings suggest that further research could validate these findings across diverse contexts, explore additional performance metrics, and assess the generalizability of the results to different types of software systems. The findings offer actionable insights for practitioners and researchers, guiding the selection of performance metrics and the application of hyperparameter tuning for improved failure prediction and test case prioritization.

Acknowledgements

This research was supported in part by IBM Center for Advanced Studies (CAS) and Natural Sciences and Engineering Research Council of Canada (NSERC) grants.

References

[1] P. Probst, M. N. Wright, and A.-L. Boulesteix, "Hyperparameters and tuning strategies for random forest," *Wiley Interdisciplinary Reviews: data mining and knowledge discovery*, vol. 9, no. 3, p. e1301, 2019.

[2] A. S. Tosun, A. Bener, and B. Turhan, "Selecting a few to get the most from many: A diversity-based active learning approach for software defect prediction," *IEEE Transactions on Software Engineering*, vol. 35, no. 5, pp. 709–722, 2009.

[3] E. Kocaguneli, T. Menzies, and B. Caglayan, "Reducing configuration overload in software product lines," in *Proceedings of the 16th International Software Product Line Conference*, 2012, pp. 181–190.

[4] I. K. M. Aydin and E. Akin, "A multi-objective artificial immune algorithm for parameter optimization in support vector machine," *Applied Soft Computing*, vol. 11, no. 1, pp. 120–129, 2011.

[5] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "The impact of automated parameter optimization on defect prediction models," *IEEE Transactions on Software Engineering*, vol. 45, no. 7, pp. 683–771, Jul. 2019.

[6] ——, "Automated parameter optimization of classification techniques for defect prediction models," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 321–332.

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

[8] A. Bertolino, A. Guerriero, B. Miranda, R. Pietrantuono, and S. Russo, "Learning-to-rank vs ranking-to-learn: Strategies for regression testing in continuous integration," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1–12.

[9] Y. Yang, Z. Li, L. He, and R. Zhao, "A systematic study of reward for reinforcement learning based continuous integration testing," *Journal of Systems and Software*, vol. 170, p. 110787, 2020.

[10] J. Chen, Y. Lou, L. Zhang, J. Zhou, X. Wang, D. Hao, and L. Zhang, "Optimizing test prioritization via test distribution analysis," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 656–667.

[11] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming google-scale continuous testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 2017, pp. 233–242.

[12] W. Zhang, Y. Yang, J. Yang, Z. Lin, H. Zhang, Y. Zou, and H. Mei, "Evasive defects: Can developers predict their own code's defectiveness?" *Empirical Software Engineering*, vol. 22, no. 4, pp. 1872–1913, 2017.

[13] E. Aydemir, P. Güler, A. Bener, and B. Turhan, "The effect of hyperparameter settings on transfer learning for automated program repair," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 294–304.

[14] N. Ramakrishnan, D. Verma, D. Gupta, U. O'Reilly, and M. Chetty, "Hybrid metaheuristics for hyperparameter tuning: An empirical investigation," in *International Conference on Artificial Intelligence and Soft Computing*, 2020, pp. 609–622.

[15] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on software engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[16] B. Bischl, M. Binder, M. Lang, T. Pielok, J. Richter, S. Coors, J. Thomas, T. Ullmann, M. Becker, A.-L. Boulesteix, D. Deng, and M. Lindauer, "Hyperparameter optimization: Foundations, algorithms, best practices, and open challenges," *WIREs Data Mining and Knowledge Discovery*, vol. 13, no. 2, p. e1484, 2023.

[17] A. Agrawal, X. Yang, R. Agrawal, R. Yedida, X. Shen, and T. Menzies, "Simpler hyperparameter optimization for software analytics: Why, how, when?" *IEEE Transactions on Software Engineering*, vol. 48, no. 8, pp. 2939–2954, 2022.

[18] A. S. Yaraghi, M. Bagherzadeh, N. Kahani, and L. C. Briand, "Scalable and accurate test case prioritization in continuous integration contexts," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1615–1639, 2022.

[19] J. Mendoza, J. Mycroft, L. Milbury, N. Kahani, and J. Jaskolka, "On the effectiveness of data balancing techniques in the context of ml-based test case prioritization," in *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2022, pp. 72–81.

[20] D. Elsner, F. Hauer, A. Pretschner, and S. Reimer, "Empirically evaluating readily available information for regression test optimization in continuous integration," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 491–504.

[21] R. Mamata, A. Azim, R. Liscano, K. Smith, Y.-K. Chang, G. Seferi, and Q. Tauseef, "Test case prioritization using transfer learning in continuous integration environments," in *2023 IEEE/ACM International Conference on Automation of Software Test (AST)*. IEEE, 2023, pp. 191–200.

[22] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Transactions on Software Engineering*, vol. 33, no. 1, pp. 2–13, 2007.

[23] R. Malhotra, "A systematic review of machine learning techniques for software fault prediction," *Applied Soft Computing*, vol. 27, pp. 504–518, 2015.

[24] D. Kühn, P. Probst, J. Thomas, and B. Bischl, "Automatic exploration of machine learning experiments on openml," *arXiv preprint arXiv:1806.10961*, 2018.

[25] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE transactions on software engineering*, vol. 28, no. 2, pp. 159–182, 2002.