2024 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW) | 979-8-3503-4479-0/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICSTW60967.2024.00014

An End-to-End Test Case Prioritization Framework using Optimized Machine Learning Models

Md Asif Khan*, Akramul Azim*, Ramiro Liscano*,
Kevin Smith† Yee-Kang Chang† Gkerta Seferi† and Qasim Tauseef †
*Department of Electrical, Computer and Software Engineering
Ontario Tech University, Oshawa, Ontario L1G 0C5
Email: {mdasif.khan, akramul.azim, ramiro.liscano}@ontariotechu.ca
†International Business Machines Corporation (IBM)
Email: smithk6@uk.ibm.com, yeekangc@ca.ibm.com
Gkerta.Seferi@ibm.com, Qasim.Tauseef@ibm.com

***Abstract*—Regression testing in software development is challenging due to the large number of test cases and continuous integration (CI) practices. Recently, test case prioritization (TCP) using machine learning (ML) has been shown to efficiently execute regression tests. This study introduces an automated, end-to-end, self-contained ML-based framework, TCP-Tune, tailored exclusively for TCP. The framework utilizes open-source version control system data to combine code-change-related features with test execution results. This integration allows the automated optimization of hyperparameters across different ML models to improve the TCP. The framework also effectively visualizes and utilizes multiple evaluation metrics to evaluate the performance of the model over several builds. Unlike existing implementations, which rely on various frameworks, TCP-Tune enables the effortless incorporation of features from multiple sources and fine-tuned models, thereby providing optimum test prioritization in the ever-changing field of software development. Our approach has helped to provide efficient TCP through experimental assessments of a real-life, large-scale CI system.**

***Index Terms*—test case prioritization, hyperparameter tuning, regression testing, machine learning**

I. Introduction

In the ever-evolving landscape of software development, integrating CI systems has become a cornerstone for ensuring the quality and reliability of software release. CI practices involve frequent and automated builds that rapidly generate vast amounts of test data. Regression testing is pivotal for identifying potential defects that may arise with each code change within a dynamic environment [1]. However, the increasing number of test cases and the need for swift feedback in CI systems pose substantial challenges for effectively executing regression testing.

Test case prioritization (TCP) has emerged as a crucial strategy for navigating this challenge, with the aim of optimize the testing process by identifying and executing critical test cases early in the testing cycle [2]. The essence of TCP lies in efficiently allocating resources to ensure thorough testing, while minimizing the time required for feedback. With a focus on large volumes of test data, TCP has become indispensable in promptly identifying failures, enhancing the overall efficiency of the software development life cycle.

Recent advancements in machine learning (ML) have ushered in a new era of TCP [3], offering promising solutions to address the complexities of prioritizing an extensive suite of test cases. ML-based approaches [4] leverage automated algorithms to discern patterns, correlations, and dependencies using features from various sources, thereby streamlining the prioritization process. However, this forces researchers to utilize multiple third-party frameworks, which often vary setting, performance, and analysis.

This paper presents a solution as an end-to-end, self-configurable hyperparameter tuning and evaluation framework called TCP-Tune, which is designed explicitly to prioritize test cases. The framework seamlessly integrates code-change histories, test execution results, and ML algorithms to optimize the hyperparameters, thereby improving the prioritization of test cases. The significance of such a framework lies in its potential to enhance the efficiency of CI systems by providing more accurate and timely feedback, reducing testing overhead, and ultimately contributing to the seamless delivery of high-quality software.

TCP-Tune enables researchers to provide the required data source, and it automatically collects, generates features, and optimizes hyper-parameters to improve the Average percentage of faults detected (APFD), a crucial metric for measuring TCP performance. The framework's ability to autonomously collect and process data, perform hyperparameter tuning, and generate insightful visualizations underscores its potential to revolutionize the TCP landscape. This study aims to comprehensively explore the functionalities of the framework, experimental validations, and imperative roles in addressing the demands of regression testing within CI systems.

The contributions of this paper are:

1) **Automated data processing**: This work contributes significantly to the introduction of stand-alone and efficient automated data preprocessing, feature generation, and engineering derived from commit histories with test execution results from the International Business Machines Corporation (IBM) Open Liberty project [5] explicitly focusing on enhancing TCP. This approach improves the efficiency of industrial systems, ensuring

979-8-3503-4479-0/24/$31.00 ©2024 IEEE
DOI 10.1109/ICSTW60967.2024.00014

that the testing process is comprehensive and resource efficient.

2) **Build-in hyperparameter tuning for TCP**: Unlike existing tuning frameworks, which focus solely on traditional ML performance measures such as precision, recall, f1-score, etc., the framework's capability to autonomously tune hyperparameters ensures adaptability and efficiency in improving TCP. Additionally, the generation of visualizations facilitates easy observation of performance trends over time, offering valuable insights into the effectiveness of tuned models.
3) **Real-world experimentation and extensive analysis**: We have shown comprehensive end-to-end implementation steps of TCP-Tune using the IBM Open Liberty dataset comprising approximately 6000 build information [5]. The TCP-Tune framework-generated graphs reveal the performance retention of the ML models over CI cycles. XGBoost consistently outperforms the other models, whereas Random Forest and Decision Trees show mixed performance. The framework's "First Failure" and "Last Failure" analyses help optimize testing strategies and resource management.

The remainder of this paper is structured as follows: Section II explores related work focusing on the frameworks used for TCP purposes, and Section III outlines the methodology, encompassing essential theoretical foundations, intricate details of TCP-Tune architecture implementations, and the hyperparameter tuning process. Section IV provides comprehensive information about the datasets and experimental procedures and scrutinizes the obtained results. Finally, Section V concludes the paper and outlines potential avenues for future research.

II. Related work

Numerous recent studies have recently aimed to enhance TCP efficiency by minimizing time and resources while prioritizing the most significant test cases for immediate execution. Several studies have incorporated ML and deep learning approaches in the field of TCP. The work by Yaraghi et al. [6] stands out as their approach involves the development of methods and frameworks to systematically collect specified features from a diverse range of 25 open-source software systems, each characterized by a sufficient number of failed builds and regression testing durations of at least five minutes. In their careful data collection process, the authors leverage various specialized frameworks and packages tailored to different aspects of CI data. They employ Bugzilla as a fault-tracking framework, PyDriller [7] for mining Git repositories to extract code change history information, Understand [8] for conducting static and dependency analysis of the source code, and RankLib [9] for training machine learning models specific to Test Case Prioritization (TCP). This multifaceted framework highlights their nuanced approach, acknowledging the complexity of CI environments and the diverse data sources inherent in the study.

Bertalino et al. [10] have shed light on two overarching strategies for ML-based prioritization: learning-to-rank and ranking-to-learn, the latter of which involves reinforcement learning. In their work, the authors introduce ten ML algorithms tailored for adoption in Continuous Integration (CI) practices, embarking on a comprehensive study to compare their efficacy. By leveraging subjects from the Apache Commons project, they explored class-level dependency graph construction as in [6] using Understand [8]. Additionally, this study employs The Weka [11] and Knime [12] frameworks for pointwise algorithms, emphasizing the versatility of framework selection based on algorithmic requirements. Notably, the RankLib [9] library comes into play for pairwise and listwise algorithms, showcasing a purposeful selection of frameworks for different ML strategies.

Tantithamthavorn et al. [13] have conducted a comprehensive study on the impact of automated parameter optimization. Their investigation, encompassing 18 datasets, reveals significant improvements in the Area Under the Curve (AUC) performance, with up to 40 percentage point enhancements. Automated optimization not only bolsters classifier stability but also substantially reshapes variable importance ranking. The study underscores that commonly used classification techniques, such as random forest and support vector machines, may need to be more optimization-sensitive. At the same time, lesser-explored methods such as C5.0, and neural networks can outperform their widely utilized counterparts post-optimization. Crucially, the authors observe that the default settings for classification techniques vary across different research frameworkkits, introducing a potential source of variation in results. For instance, default settings for the number of decision trees in a random forest classifier differ across packages such as bigrf [14], MATLAB [15], Weka [11], and random forest. This variation in default settings emphasizes the need for a nuanced exploration of the parameter space, particularly for parameter-sensitive techniques, shedding light on the importance of consistent and meticulous configurations in defect prediction studies.

Scikit-learn [16] is a widely used machine learning library that offers various frameworks for various tasks, including classification, regression, and clustering. However, it does not explicitly focus on test case prioritization. TCP-Tune is developed to address this issue by introducing performance measures such as APFD and Average Failure Position (AFP) to improve the order of test case execution. The framework also automates hyperparameter tuning techniques for APFD improvement. CARET [17], an R package, simplifies the process of training and evaluating predictive models but lacks a specific focus on test case prioritization. TCP-Tune fills this gap by providing specialized metrics and automation techniques for test case prioritization, enabling developers to assess and enhance the efficiency of their test case execution order, ultimately improving fault detection efficiency in software testing processes.

Hence, the implementation of state-of-the-art methodologies often relies on diverse frameworks for data collection within CI environments. These frameworks are crucial for gathering information on code changes, test results, and other relevant

metrics. However, a noteworthy observation is the absence of a dedicated, singular framework designed explicitly for the nuanced TCP task. Despite the plethora of available frameworks catering to different facets of the CI, the need for a unified framework for TCP underscores the gap in the current framework ecosystem. This gap becomes particularly pronounced because TCP requires a specialized approach to optimize the testing processes efficiently.

III. TCP-Tune Architecture

Our framework has three significant modules for automating the prioritization process: (1) feature processing module (*M1*), (2) machine learning module (*M2*), and (3) evaluation module (*M3*) as shown in Figure 1.

To showcase the framework architecture, we chose a dataset from the IBM Open Liberty framework, a fast and composable Java-based application server, to create applications and cloud-based services [5]. Open Liberty is an open-source licensed version. The Functional Acceptance Test (FAT) is the core part of IBM testing and differs from unit tests because FAT tests are executed on a running Open Liberty image. The dataset contains approximately 4.6 million test execution results from 6,000 builds, with approximately 1% failure.

A. Feature Processing Module

Our framework requires two distinct features: code-change-related features (residing in Github for open-source projects) and test-execution-related features (collected after test executions in a successful build) as shown in Figure 2 (*Data Collection*). The models use these features to train the hyperparameters to obtain the best-prioritized test case list and report the TCP performance over different builds.

1) ***Code-change-related features****:* TCP-Tune can analyze and process any open source project from the GitHub repository's commit history, performing calculations related to test cases executed in the builds. It retrieves information about the changed file names from either a local GitHub repository's commit history or makes API calls to GitHub. Subsequently, the commit hash, commit time, and change files are extracted by initializing a dictionary. It updates the dictionary with the total number of lines inserted, deleted, and changed in all the files for a given commit object.

2) ***Test-execution-related features****:* Test execution data can be collected using various means. TCP-Tune requires the test execution results as CSV files. However, the data involves cleaning and preprocessing. In that case, it can filter out specific builds that are not part of usual test executions, match particular patterns in the child build information to remove respin builds, and remove builds with missing details. Some of the notable features are **Failed** (denotes whether the test execution has passed or failed), **Test total** denotes the number of test cases in a given test suite, **duration**, **start time** and related Build and Job information.

3) ***Data Integration****:* One of the critical steps of TCP-Tune is combining CI test results and Git commits information and filtering and merging data related to job names, commit references, and other attributes. Figure 2 (*Feature Mapping*) shows how the framework utilizes CI build information and related commit head data to calculate various statistics related to commits, lines inserted/deleted, and changed files. Finally, we use nine features to represent the test suites as feature vectors to train our ML algorithms. The features in our study include the **Test suite ID**, that is, the name of the test suite, which is converted into a unique integer ID for use as a feature, and **Test total**. Additionally, we consider the number of **files changed**, **lines deleted**, and **lines inserted** in a particular commit as discrete values. Boolean features such as **contains unit** (which means the file changes include a unit test), **contains fat** (which means the file changes include a fat test), **contains java** (which means the file changes include a Java file), and **Match count** (which means the changed file contains the test suite name) can indicate whether specific characteristics are present or absent in file changes. These features collectively contribute to the evaluation of TCP using ML.

B. Machine Learning Module

1) ***Machine Learning Models****:* TCP-Tune focuses on tree-based machine learning models, including Extreme Gradient Boosting (XGBoost), Gradient Boosting (GBoost), Random Forest (RF), and Decision Trees (DT). These models are chosen for their adaptability, interpretability, and ability to handle diverse feature types, aligning with prior research [3], [6], [10] emphasizing the effectiveness of tree-based models in software testing. The features are provided to the ML models to predict whether the test suite will pass or fail.

2) ***Hyperparameter optimization****:* In TCP-Tune, the hyperparameter tuning process is guided by state-of-the-art findings and domain-specific knowledge. A time budget threshold of 24 hours is set for the tuning process in accordance with Kuhn's concept [18]. However, this framework differs from previous works [6], [10], [19] by considering an extended set of hyperparameters, contributing to a more comprehensive optimization of machine learning models.

A baseline model training phase is initiated to establish a performance benchmark for models in their default configurations. Key metrics, such as precision, recall, F1-score, and AUC, are recorded to quantify the initial capabilities of the models before hyperparameter adjustments.

TCP-Tune conducts hyperparameter tuning using a grid search method, systematically exploring all possible configurations within defined parameter spaces. The process involves evaluating the classifier performance for each configuration, considering subsets of the training dataset, and assessing the model performance on data not included in the training subset.

The RF model was tuned with parameters including n_estimators, criterion, and max_features, whereas the DT model's hyperparameters were adjusted for max_depth, criterion, splitter, and max_features. XGBoost was optimized with parameters such as subsample, eta, and max_depth, whereas GBoost utilized hyperparameters including learning_rate, n_estimators, and max_depth.The user can add,

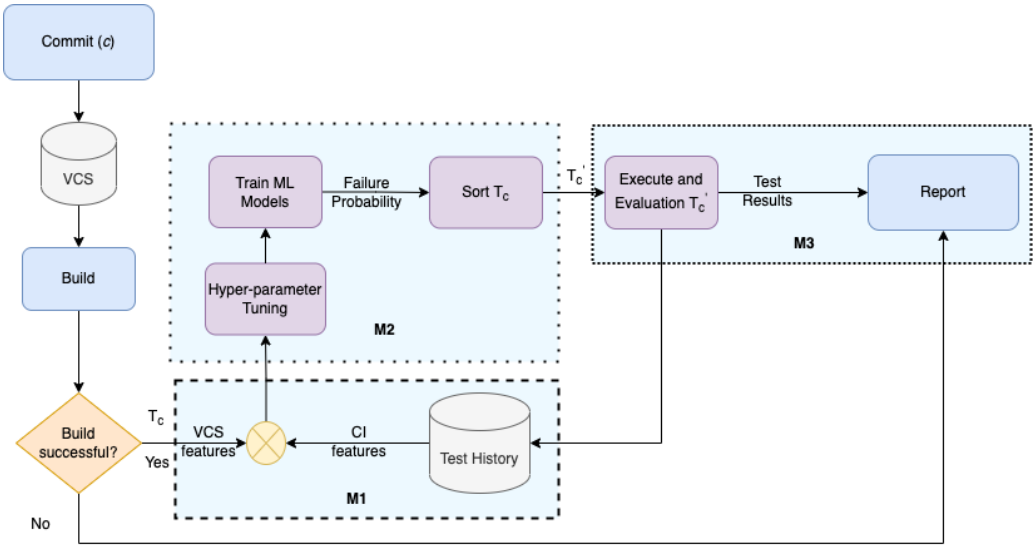


Fig. 1. Overview of automated modules of TCP-Tune framework

update the hyperparameter based on their requiremet. This framework considers every combination of hyperparameters and train related ML models. After each training the model are tested and their evaluation metrics are calcualted. Following hyperparameter optimization, the models are further employed for TCP, generating prediction probability values for the test suites. The test suites are then prioritized based on these probabilities, with a higher priority given to suites deemed more likely to uncover faults. The subsequent sequential execution of test suites aims to expedite fault detection and relevant metrics, including APFD, are calculated for evaluation purposes.

Tuning the hyperparameter can add overhead to the TCP process. To mitigate this issue, TCP-Tune allows developers to implement TCP without tuning hyperparameters. Based on project requirements, developers can choose the frequency of hyperparameter tuning. On the other hand, we know that other hyperparameter tuning techniques exist, such as random search or Bayesian optimization, which are faster than grid search as they only search some combinations. The current framework helps us to use grid search as the base model, and future work will include other techniques.

*3) **Test Case Prioritization**:* In the subsequent phase, fine-tuned models are utilized to generate prediction probability values for each test suite. Subsequently, these prediction probabilities are employed to arrange the test suites in descending order, prioritizing those with the highest probability of failure at the forefront of the list. This strategic sorting ensures that test suites are more likely to uncover faults or failures in the software take precedence, with those exhibiting lower probabilities following suit. The underlying rationale is to streamline the execution process, focusing on test suites deemed most effective in identifying potential software issues.

Following the established order, the test suites are then executed sequentially, commencing with those positioned at the top of the list. This sequential execution methodology is deliberately crafted to expedite the detection of potential software issues by systematically exploring prioritized test suites. Subsequent to the execution of the test suites, a thorough analysis is conducted, involving the calculation of essential metrics and AFP graph generation.

C. Evaluation Module

The evaluation phase serves as a comprehensive assessment of the effectiveness of fine-tuned models in enhancing the TCP strategies.

*1) **Average percentage of faults detected (APFD)** [20]–[22]:* Let T be a test suite consisting of n test cases, and Rothermel et al. [23] calculated the APFD of a prioritized test suite T as follows:

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

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. [21] plotted the executed test suite rate and detected the fault rate along the x and y-axis, to calculate the APFD. The minimum APFD value is zero, whereas the maximum value is one. Higher APFD values indicate that the faults are identified faster using fewer test suites.

*2) **Average Failure Position (AFP)**:* We have introduced new metrics to evaluate the effectiveness of prioritized test suites. This is known as the *Average Failure Position* (AFP). The formula used is as follows:

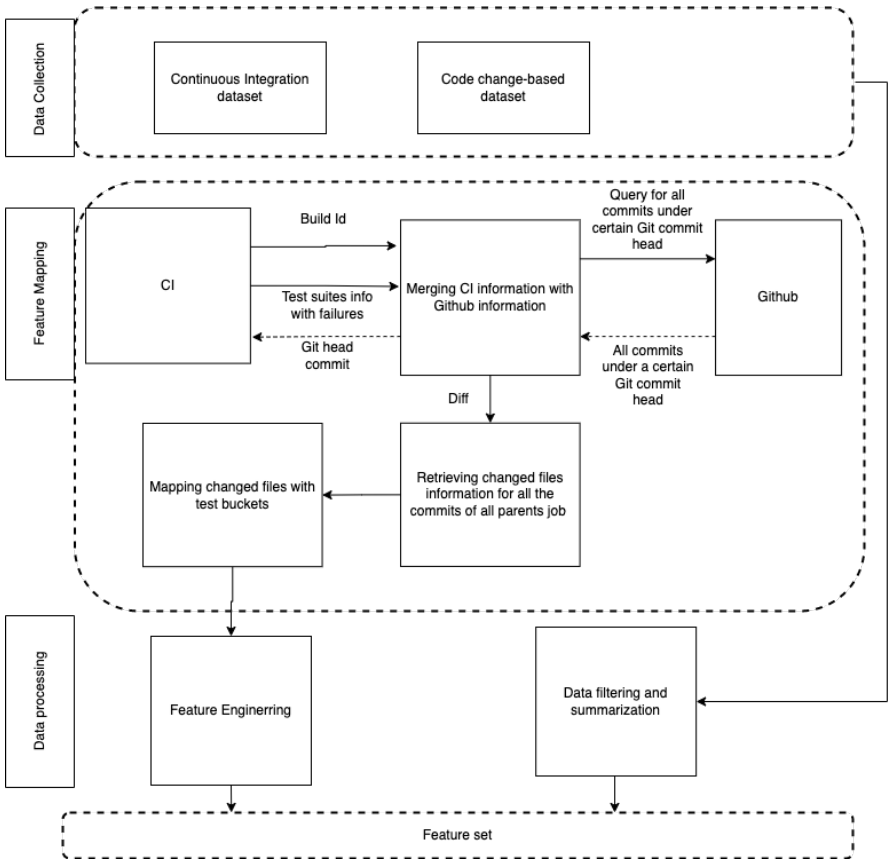


Fig. 2. Overview of feature processing module of TCP-Tune framework

$$AFP = \frac{1}{n}\sum_{i=1}^{n} x_i = \frac{1}{n}\left(x_1 + \cdots + x_n\right) \tag{2}$$

Where,

- AFP = Average Failure Position
- x_i = Failure position to be averaged

*3) **Precision, Recall, F1-Score, and AUC**:* These metrics [24] provide a comprehensive assessment of the model performance, including precision (how many predicted failures are correct), recall (how many actual failures are detected), F1-Score (harmonic mean of precision and recall), and AUC [2], [25]. These metrics are crucial for understanding the model's effectiveness in test case prioritization and provide a well-rounded view of the TCP performance.

IV. Results Analysis

The TCP-Tune framework is used to analyze the execution of test suites across approximately 6,000 different CI builds and their related code-change-related features using the IBM Open Liberty dataset [5]. After the feature generation phase, the framework trains ML models with individual features one by one and then reports the results using all the features combined. Then, the feature importance module provides us with the feature importance of the nine features obtained; tunes the hyperparameters of the four ML models, and reports the APFD and AFP values along with tuning time. Finally, the framework generates graphs to show the TCP performance over several builds. The entire process demonstrates how TCP-Tune can be utilized to optimize ML models for better TCP scores.

A. *Feature Processing Module Analysis*

In this experiment, we use TCP-Tune to assess the impact of individual features on model performance and compare with the models trained with all the features. We record four evaluation metrics for the four machine learning models. Table I shows only the maximum values obtained during this process. For example, the maximum precision value when trained with all features generated by the XGBoost classifier is 0.9392, where DT trained with single features generates a maximum value of 0.9375.

Subsequently, our focus shifts toward feature selection, which is a critical aspect for optimizing TCP models. We determine the best feature set from the RF model by considering a feature importance measure called the Mean Decrease in Impurity (MDI). The MDI measures the average reduction in impurity across decision trees in an ensemble, indicating the importance of a feature in a random forest model. TCP-Tune analyzes the feature set for selection using MDI and shows their results are shown in Figure 3. The number of test cases in test suites has the highest feature importance, followed by

TABLE I
ML CLASSIFIERS PERFORMANCE MEASURE (MAX) VS INDIVIDUAL FEATURES

Precision		Recall		F1- score		AUC	
All	Individual	All	Individual	All	Individual	All	Individual
0.9392 (XGB)	0.9375 (DT)	0.5898 (RF)	0.0547 (RF)	0.7154 (RF)	0.0973 (RF)	0.7948 (RF)	0.5272 (RF)

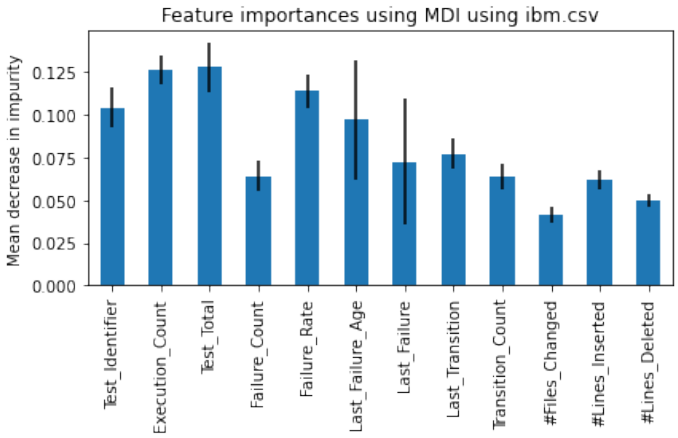


Fig. 3. Feature importance generated by TCP-Tune framework using MDI

the number of times the test suites are executed. On the other hand, the number of files changed in the commit information has the most negligible impact on ML performance. We can further shorten the feature list to reduce the feature set while maintaining the same performance as in the study by Yaraghi et al. [6], who only selected the top 15 features out of 150 features based on the information gained from the RF model.

B. ML Module Analysis

In this experimental investigation, TCP-Tune reports the impact of hyperparameter tuning on the failure prediction capabilities of ML models. TCP-Tune reports critical performance metrics, including precision, recall, F1-score, and AUC. The results of hyperparameter tuning demonstrate a substantial increase in the performance metrics of the ML models. Precision increased from 0.9392 in the default model to a perfect score of 1.0000 after tuning, indicating a noteworthy improvement of 6.49%. Similarly, recall experienced a positive shift from 0.5898 to 0.6160, reflecting a 4.46% increase. While the F1-score exhibited a more modest increase of 0.98%, achieving 0.7225 after tuning, it still contributes to the overall improvement in predictive capabilities. However, it is crucial to note the observed decrease in the AUC from 0.7948 to 0.6254, representing a 21.31% decline. This discrepancy emphasizes the trade-offs inherent in hyperparameter tuning, where changes in others may counterbalance improvements in specific metrics. Overall, the positive adjustments in precision and recall highlight the effectiveness of hyperparameter tuning in fine-tuning the model to achieve a more balanced and accurate prediction, even though managing the trade-offs between different performance metrics is challenging.

The considerable performance discrepancies observed across different hyperparameter combinations in Table II raise concerns regarding potential overfitting or underfitting. This highlights the necessity for a refined approach to fine-tuning hyperparameters. For instance, achieving a flawless precision score for DT with specific hyperparameters might signal overfitting; because the model could be overly complex, capturing noise as significant patterns. These scenarios warrant further investigation using the TCP-Tune framework.

TABLE II
ML CLASSIFIER PERFORMANCE (MAX) BEFORE AND AFTER HYPERPARAMETER TUNING [26]

Metric	Default	Tuned	Change (%)
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%

On the other hand, TCP-Tune's analysis reveals insights into the hyperparameter tuning costs for diverse ML models. The DT exhibits low tuning times, escalating with higher `max_depth` values. RF tuning times significantly increase with `n_estimators`, ranging from seconds to hours. GBoost shows varied tuning times based on the `max_depth`, `n_estimators`, and learning rate (`LR`). XGBoost generally demands higher tuning times, particularly with a larger `max_depth`, peaking at `max_depth=5,000`. Ensemble approaches, such as Random Forest (RF) and XGBoost, require significant computational resources dictated by hyperparameters. This highlights the need to balance the complexity of the model with its computational cost. Effective tuning procedures are essential for balancing optimal performance and resource management.

C. TCP Performance Analysis

Using optimized ML models, TCP-Tune thoroughly analyzes the impact of hyperparameter tuning on APFD values, as illustrated in Figure 4. The performance of each hyperparameter combination is systematically evaluated. DT demonstrates mixed performance, with a minimum APFD (mean) of 0.5069, indicating potential inefficiencies in test prioritization, yet a notable maximum APFD of 0.9791, highlighting its effectiveness in identifying fault-prone test cases. RF exhibits

consistent and robust test case prioritization, ranging from a minimum APFD (mean) of 0.5242 to a maximum of 0.9804. GBoost shows a broader range of APFD values, whereas the XGBoost Classifier demonstrates highly consistent performance, with a minimum APFD (mean) of 0.6567 and a maximum of 0.9835. A comparison of the mean APFD values between randomly selected hyperparameter models (blue line) and hyperparameter-tuned models (red line) across all ML models underscores the significant improvement achieved through systematic hyperparameter tuning. This analysis underscores the crucial role of hyperparameter optimization in enhancing ML model performance, advocating for systematic approaches such as grid search for optimal results in model development.

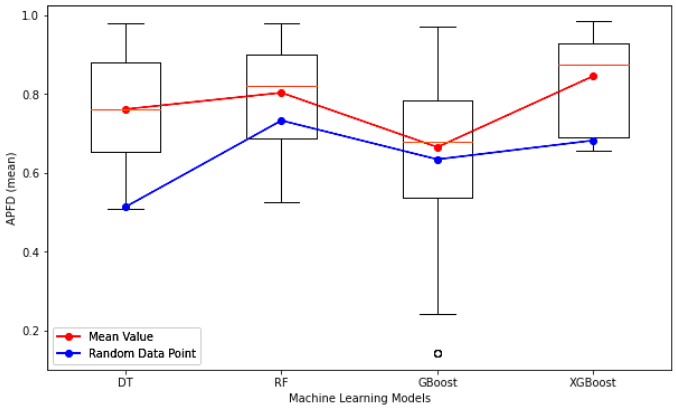


Fig. 4. Change in APFD value for different hyperparameter combinations of ML models [26]

D. *TCP Visualization Analysis*

We employ the TCP-Tune framework to assess the quality of ML model performance over a specific period. Our framework-generated visualizations serve as a crucial resource for gauging whether the performance of particular ML models is undergoing degradation, necessitating retraining using the recent data. This emphasizes the inherent need for visualization frameworks that enable real-time tracking of the ML model performance, a feature not commonly provided by current state-of-the-art solutions. The availability of such visual insights contributes significantly to maintaining the optimal model efficacy and addressing performance fluctuations over time. The AFP graph in Figure 5 shows that the best-performing ML algorithm over the selected CI cycles is XGB, as shown in Figure 5(d). The blue line, which represents the AFP values of XGB, consistently produces a lower AFP values than the actual test suite execution. RF is the worst performing ML algorithm over the selected CI cycles, as shown in Figure 5(b). On the other hand, DT and GBoost, as shown in Figures 5(a) and 5(c), show the AFP value of the actual CI cycles; there are few instances where the AFP value of these two ML algorithms are higher.

We further extend our evaluation by examining two other factors using the TCP-Tune framework, as shown in Figure 6:

1) **First Failure:** This represents the position at which the first failure is located. Based on this requirement, the tester may decide to halt further execution, thus saving response time and resources. Figure 6 shows that for the GBoost classifier the first failure is predominantly found under the 200 test execution indexes.
2) **Last Failure:** This represents the location of the last failure. In comparison, the tester may not be confident of the last failure. However, it can gauge how a particular ML algorithm performs in terms of APFD by examining at the last failure list of a consecutive prioritized list. Figure 6 shows that for the GBoost classifier the last failure is predominantly found between 500-700 test execution indexes.

Extensive model performance reporting and visualization underscores the practical applicability of the proposed frameworks and the potential to integrate the developed framework into real-world CI testing scenarios.

V. Conclusion

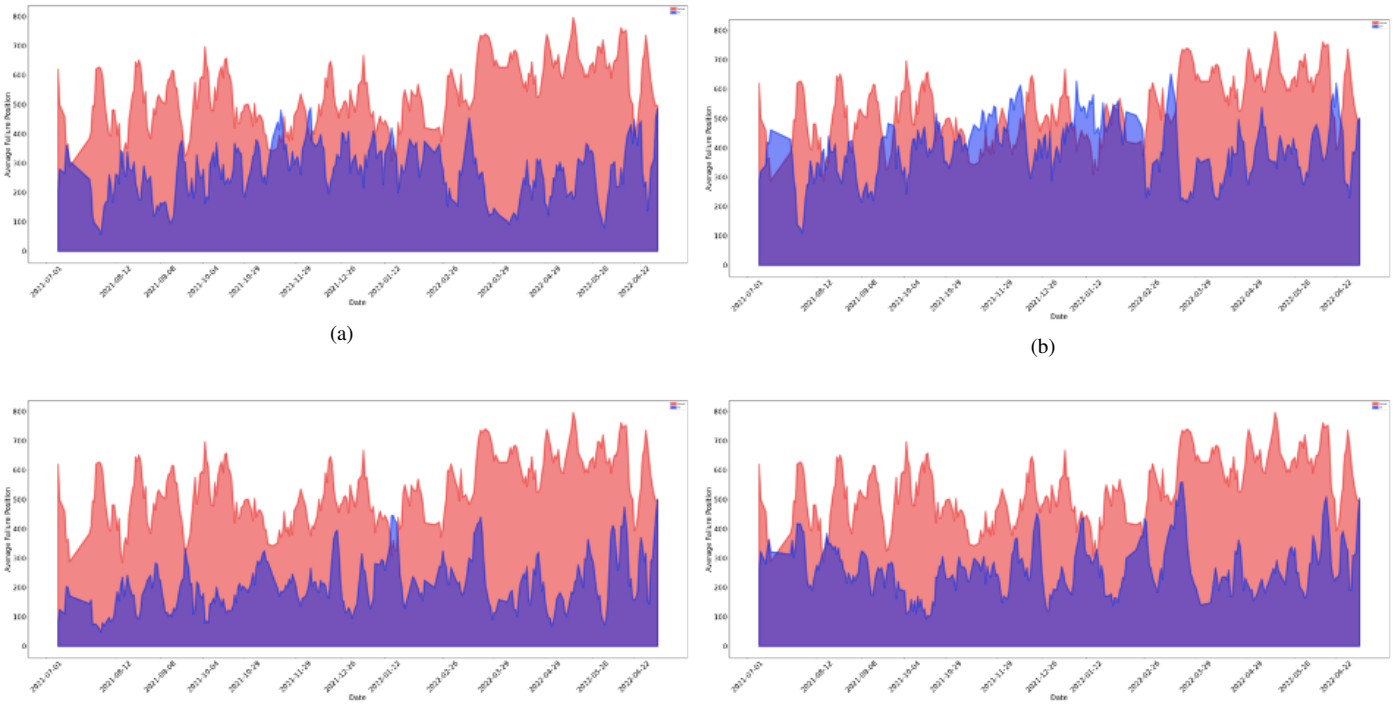
This study introduces TCP-Tune, a novel end-to-end, self-contained hyperparameter tuning and a performance visualization framework. It introduces an automated process for integrating commit histories with test execution results, thereby improving the effectiveness of TCP in industrial systems. It also introduces an end-to-end hyperparameter tuning framework for TCP, that autonomously tunes hyperparameters, ensuring adaptability and efficiency. The framework further includes visualizations for observing performance trends over time. Our study extends beyond theoretical propositions through real-life experimentation with a large-scale continuous integration dataset, thereby demonstrating the practical applicability of the proposed approach. However, our study was validated using only one large-scale CI dataset and four ML models. Future studies will explore its relevance using a broader range of ML models in the production environment.

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] D. Suleiman, M. Alian, and A. Hudaib, "A survey on prioritization regression testing test case," in *2017 8th International Conference on Information Technology (ICIT)*. IEEE, 2017, pp. 854–862.
[2] J. A. P. Lima and S. R. Vergilio, "Test case prioritization in continuous integration environments: A systematic mapping study," *Information and Software Technology*, vol. 121, p. 106268, 2020.
[3] R. Malhotra, "A systematic review of machine learning techniques for software fault prediction," *Applied Soft Computing*, vol. 27, pp. 504–518, 2015.
[4] R. Lachmann, "12.4-machine learning-driven test case prioritization approaches for black-box software testing," *Proceeding-ettc2018*, pp. 300–309, 2018.
[5] "Websphere liberty - overview." [Online]. Available: https://www.ibm.com/ca-en/cloud/websphere-liberty
[6] 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.



(a) (b) (c) (d)

Fig. 5. AFP graph using (a) DT (b) RF (c) GBoost (d) XGB

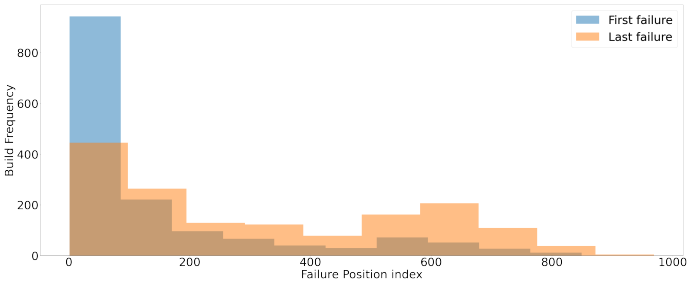


Fig. 6. Distribution of First and Last failure position using GBoost

[7] G. Gousios, "The ghtorent dataset and tool suite," in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 233–236.

[8] Scientific Toolworks, Inc., "Understand™," https://www.scitools.com/, 2020, retrieved March 8, 2024.

[9] (2024) RankLib. https://sourceforge.net/p/lemur/wiki/RankLib/.

[10] 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.

[11] Weka. https://www.cs.waikato.ac.nz/ml/weka/.

[12] KNIME. https://www.knime.com/knime-software.

[13] C. Tantithamthavorn, A. E. Hassan, and K. Matsumoto, "The impact of class rebalancing techniques on the performance and interpretation of defect prediction models," *IEEE Transactions on Software Engineering*, 2018.

[14] A. Lim, L. Breiman, and A. Cutler, "bigrf: Big random forests: Classification and regression forests for large data sets," *URL http://CRAN. R-project. org/package= bigrf. R package version 0.1-11*, 2014.

[15] V. Matlab, "8.5. 0 (r2015a)," *The MathWorks Inc., Natick, MA*, 2015.

[16] scikit-learn. https://scikit-learn.org/stable/.

[17] caret. https://www.nitrc.org/projects/caret/.

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

[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] M. A. Khan, A. Azim, R. Liscano, K. Smith, Y.-K. Chang, S. Garcon, and Q. Tauseef, "Failure prediction using machine learning in ibm websphere liberty continuous integration environment," in *Proceedings of the 31st Annual International Conference on Computer Science and Software Engineering*, 2021, pp. 63–72.

[21] 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.

[22] H. de S. Campos Junior, M. A. P. Araújo, J. M. N. David, R. Braga, F. Campos, and V. Ströele, "Test case prioritization: a systematic review and mapping of the literature," in *Proceedings of the XXXI Brazilian Symposium on Software Engineering*, 2017, pp. 34–43.

[23] 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.

[24] H. Jahan, Z. Feng, S. Mahmud, and P. Dong, "Version specific test case prioritization approach based on artificial neural network," *Journal of Intelligent & Fuzzy Systems*, vol. 36, no. 6, pp. 6181–6194, 2019.

[25] A. T. Njomou, A. J. B. Africa, B. Adams, and M. Fokaefs, "Msr4ml: reconstructing artifact traceability in machine learning repositories," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 536–540.

[26] M. A. Khan, A. Azim, R. Liscano, K. Smith, Q. Tauseef, G. Seferi, and Y.-K. Chang, "Machine learning-based test case prioritization using hyperparameter optimization," in *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST '24)*, ser. AST '24. ACM, 2024, p. 11. [Online]. Available: https://doi.org/10.1145/3644032.3644467