Volume: 15 Issue: 06 Pages: 6218 – 6228 (2024) ISSN: 0975-0290

Machine Learning for Test Case Prioritization in Continuous Integration: A Comprehensive Analysis

Hemant Kumar
Department of Computer Science, Babasaheb Bhimrao Ambedkar University, Lucknow 226025, India
Email: hemant20192@gmail.com
Vipin Saxena
Department of Computer Science, Babasaheb Bhimrao Ambedkar University, Lucknow 226025, India
Email: profvipinsaxena@gmail.com

ABSTRACT

This study introduces an innovative Predictive Test Prioritization (PTP) methodology for Continuous Integration (CI), utilizing historical test case execution data. To predict the probability of success or failure in new test cases, machine learning classifiers like k-nearest Neighbors, Random Forest, Support Vector Machine (SVM), Gradient Boosting, and Logistic Regression are applied and trained on historical data. The evaluation process encompasses metrics such as F1-score, recall, accuracy, and precision, offering a nuanced understanding of the effectiveness of the classifiers. The overarching goal is to optimize test prioritization, potentially enhancing software testing efficiency. This research offers valuable insights into continuous integration systems, emphasizing the pivotal role of predictive strategies in refining testing practices and contributing to the knowledge base in CI.

Keywords – **Continuous Integration, Machine Learning, Predictive Approach, Software Testing, Test Prioritization.**

Date of Submission: April 03, 2024 Date of Acceptance: April 30, 2024

I. INTRODUCTION

In the realm of contemporary software development, CI stands as a linchpin, orchestrating seamless code integration and facilitating the rapid delivery of top-tier software. As software projects burgeon, enveloping intricate code repositories and accelerating development cycles, the significance of an astute test prioritization strategy becomes increasingly pronounced. This research probes the nuances of test prioritization within modern CI environments, where sprawling codebases necessitate a nuanced and sophisticated approach. In the dynamic landscape of CI, characterized by automated code integration, a resilient testing strategy is indispensable. However, the impracticality of executing the entire test suite for each integration, compounded by the burgeoning size of codebases and the demand for expeditious development cycles, underscores the strategic imperative of test prioritization. This approach empowers development teams to focus on critical test cases, optimize resource allocation, and expedite the feedback loop. Despite the acknowledged significance of test prioritization, conventional methods grapple with substantial challenges, often relying on historical data or static rules that lack adaptability to evolving software projects. Inefficient prioritization poses risks such as suboptimal resource utilization, delayed issue detection, and diminished overall CI process effectiveness. This research tackles the challenge head-on by proposing an innovative and predictive approach, departing from static heuristics. By advocating for the integration of machine learning techniques to forecast the impact of test cases based on historical execution data, this approach introduces a paradigm shift in test prioritization. By championing innovation, this approach empowers development teams to make informed decisions about test prioritization, streamlining the testing process, and ensuring the timely identification of critical issues. Through a systematic exploration of these objectives, this paper contributes valuable insights to the ongoing discourse on optimizing software testing practices in CI environments. Emphasizing the intersection of machine learning and test prioritization strategies, it underscores the need for adaptive and intelligent approaches to address the challenges posed by traditional methods.

This research aims to pave the way for a more efficient and dynamic testing landscape in the ever-evolving realm of software development.

II. RELATED WORK

This review investigates recent machine learning research in testing methodologies and continuous integration, specifically focusing on predictive test prioritization techniques. The present approach automates and enhances prioritization, demonstrating efficiency within CI. Positioned within the landscape of predictive methods, the research emphasizes the critical role of techniques which play vital role in optimizing software testing practices, particularly for new test cases based on historical data. Let us describe some of the important references.

In the year 2019, Durelli et al. [1] conducted a systematic mapping study to review the state-of-the-art applications of Machine Learning (ML) in automating software testing. The study involved selecting 48 primary studies and categorizing the studies based on study type, testing activity, and ML algorithm. Findings revealed that ML algorithms were predominantly used for test case generation, refinement, and evaluation, as well as in test oracle construction and predicting testing-related costs. The study concluded by outlining commonly used ML algorithms and emphasized the need for more empirical studies to better understand the application in automating software testing activities. Speiser et al. [2] utilized random forest classification, a widely adopted machine learning method, to construct prediction models across diverse research domains. The study focused on optimizing variable selection within the framework to reduce the number of variables for enhanced model efficiency. Using 311 online classification datasets, the researchers assessed prediction error rates, variables, computation times, and the area under the receiver operating curve. The comparison covered diverse dataset types and methods, highlighting Jiang's method and the VSURF R package for optimal performance in most cases. For datasets with numerous predictors, varSelRF and Boruta were recommended for computational efficiency, contributing valuable insights for tailored applications in expert and intelligent systems. Zhu et al. [3] introduced a data mining-based diabetes prediction model that integrated Principal Component Analysis (PCA), K-means clustering, and logistic regression. The study addressed challenges related to the sensitivity of K-means clustering to initial positions, impacting subsequent logistic regression performance. Utilizing the Pima Indians Diabetes (PID) dataset, the model sequentially applied PCA to enhance K-means clustering and logistic regression for classification. Results demonstrated the model's effectiveness, with PCA improving clustering and surpassing other studies. Notably, the K-means output achieved 25 more correctly classified data points, accompanied by a 1.98% increase in logistic regression accuracy. Practical validation in healthcare settings involved predicting diabetes using electronic health records, and further experiments confirmed the model's versatility beyond the Pima Indians diabetes dataset.

In the year 2020, Marijan et al. [4] addressed challenges in machine learning software testing, recognizing its widespread application. The study emphasized machine learning's vulnerability to deception, requiring scrutiny, especially in safety-critical contexts. The authors advocated software verification and testing as crucial techniques, emphasizing error detection, specifically, highlighted extensive testing challenges, providing insights into six key areas and current approach limitations. The study not only identified challenges but also proposed a research agenda to advance machine learning testing, contributing to the field's state-of-the-art. The insights offered valuable guidance for addressing complexities, ensuring correctness, and enhancing trustworthiness in machine learning applications. Yucalar et al. [5] addressed the challenges of manually predicting software defects in large and complex projects, proposing automated predictors to identify faulty modules efficiently. The paper advocated for improved fault predictors, suggesting that combining base predictors through ensemble strategies enhances their fault-detection performance. The study empirically compared ten ensemble predictors to baseline predictors, using 15 software projects from the PROMISE repository. Evaluation metrics included F-measure (FM) and ROC-AUC Curve, revealing that ensemble predictors demonstrated a notable improvement in fault detection. The research contributed valuable insights to software quality engineering, emphasizing the practical application of ensemble predictors to enhance software fault prediction in real-world projects. Meçe et al. [6] conducted a review on the application of machine learning in Test Case prioritization (TCP) for regression testing. The study explored recent methodologies, techniques, and outcomes in the domain. Machine learning techniques were employed in TCP to enhance efficiency, with various studies using metrics to measure effectiveness. The application of machine learning in TCP demonstrated promise, offering insights into optimizing test case prioritization in software engineering. Braiek et al. [7] explored ML program testing in safety-critical systems, integrating software testing principles like code coverage, mutation testing, and property-based testing. The methodology included a thorough review, outlining inherent challenges and presenting techniques from the literature. The results compiled existing practices, exposed gaps in ML program testing literature, and provided insights that guided future research directions, offering valuable recommendations for addressing evolving demands in ML testing.

In the year 2021, Khatibsyarbini et al. [8] investigated the utilization of ML in TCP to enhance software testing performance. The paper underscored the importance of TCP in addressing resource and time challenges associated with the multiple phases of software testing and focused on ML techniques in TCP, the review was based on specific research questions and aligned methodology, involving the analysis of 110 studies (58 journal articles, 50 conference papers, and 2 other articles). The findings highlighted the growing trend of ML techniques in TCP, while acknowledging the need for improvement. The review emphasized the diverse characteristics of ML techniques and significant role in TCP for software testing.

In the year 2022, Goyal and Sinha [9] conducted a review on software defect-based prediction using logistic regression, addressing challenges in software testing. The study explored the development of automated predictors analyzing errors through various learning methods. It emphasized the need for efficient machine learning-based prediction systems. The review compared studies, detailing measurement methods, challenges, and system effectiveness. Findings indicated the use of 44% of NASA's PROMISE data, 68.18% software metrics, and a 16%

utilization of Logistic Regression. Gezici and Tarhan [10] explored explainable AI for software defect prediction using a Gradient Boosting (GB) classifier. The study addressed the increasing interest in explainability for stakeholders using AI and machine learning software, emphasizing the need to comprehend predictions from black-box AI-based systems. Applying post-hoc model-agnostic methods—ELI5, LIME, and SHAP—to an SDP dataset from NASA, the research aimed to enhance GB classifier explainability. The results demonstrated a consistent and model-agnostic approach to quantify explainability, with ELI5, LIME, and SHAP providing coherent explanations for the GB model. Omri [11] minimized software testing costs through a data-driven approach, addressing limitations in existing techniques. The proposed method, using software quality and code churn metrics, demonstrated high accuracy on automotive software at Daimler. The thesis introduced a test case prioritization model based on code change features, test execution history, and component development, reducing CI costs by predicting triggering test suites. For capturing domain knowledge and tester preferences, a test case execution scheduling model with a probabilistic graph solved the optimal test budget allocation problem online in CI cycles and offline during release planning. The theoretical cost model, validated on energy management and predictive maintenance applications, reported over 95% of test failures while executing only 43% of available test cases. Pan et al. [12] systematically reviewed ML based TSP for optimizing regression testing in the context of CI. The analysis of 29 studies from 2006 to 2020 addressed five research questions, providing insights for categorizing future TSP studies and highlighting key aspects in the past ML-based TSP landscape. Yaraghi et al. [13] addressed the need for efficient regression testing in CI to maintain software quality without causing significant delays. They explored TCP techniques using ML to handle the dynamic nature of CI. The authors conceptualized a data model and defined a comprehensive set of features for 25 open-source software systems, focusing with sufficient failed builds and regression testing durations of at least five minutes. The collected dataset was used to answer research questions related to data collection time, the effectiveness of ML-based TCP, the impact of features on effectiveness, the decay of ML-based TCP models over time, and the trade-off between data collection time and the effectiveness of ML-based TCP techniques. Da Roza et al. [14] introduced a TCP approach for CI environments, using the sliding window method with various ML algorithms and tailored for CI constraints like test budget and case volatility, the TCP method applied Reinforcement Learning (RL) principles. The Random Forest (RF) algorithm and a Long Short Term Memory (LSTM) network were proposed as RL alternatives, emphasizing simplicity and efficiency. The study, with three time budgets and eleven systems, demonstrated the approach's applicability in prioritization time and duration between CI cycles. RF outperformed RL in more restrictive budgets, achieving the best NAPFD values in approximately 72% of cases, while the LSTM network performed well in 55% of cases across all systems and budgets. The results highlighted the efficiency and effectiveness of the RF algorithm, prompting discussions on the implications for the evaluated algorithms' usage.

In the year 2023, Marijan [15] compared machine learning-based test case prioritization in continuous integration testing, addressing time constraints and data abundance. Emphasizing machine learning's potential in handling complex testing challenges, especially in continuous integration, where large datasets resulted from iterative code commits and test runs, the research focused on training predictors to identify test cases for expedited regression bug detection during code integration. Various machine learning approaches were evaluated with real-world and augmented industrial datasets, considering factors like continuous integration time budget and the length of test history for training classifiers. Results highlighted varied model performance based on test history size and time budgets, emphasizing the importance of configuring machine learning approaches for optimal test prioritization in continuous integration testing. Sánchez-García et al. [16] explored the application of Gradient Boosting (GB) machine learning regression algorithm optimized through Differential Evolution (DE) for Software Testing Effort Prediction (STEP). The study compared the prediction accuracy of GB-DE with GB optimized through Particle Swarm Optimization (PSO) and Genetic Algorithms (GA). Additionally, the performance of GB-DE, GB-PSO, and GB-GA was compared to that of statistical regression (SR). Seven datasets from an international public repository for software projects were used for evaluation. The results demonstrated that GB-DE outperformed SR in all seven datasets at a 95% confidence level, while GB-PSO and GB-GA performed better than SR in four and three datasets, respectively. The study concluded that GB-DE was suitable for STEP in new or enhancement projects developed in either the third or fourth programming language generation. Ramesh et al. [17] developed the Educational Assistant for Software Testing (EAST) framework to enhance students' software testing skills using digital teaching methods such as Computer Assisted Instruction (CAI). The framework incorporated Natural Language Processing (NLP), machine learning, and information retrieval techniques. The study introduced a novel approach using a Group Search Optimized two-stage hybrid Support Vector Machine-K-Nearest Neighbor (SVM-KNN) classifier to analyze parameters introducing bugs in bug reports. The Group Search Optimization (GSO) algorithm addressed data sparsity by optimizing parameter selection for the hybrid classifier. Two bug report datasets were utilized for testing, collected from an open-source community and mobile application development companies. Experimental results demonstrated that the EAST framework could improve outdated teaching methodologies based on various performance metrics. Kumar and Saxena [18] combined deep neural networks with SVM, RF, and XGBoost ensemble models for cross-project software defect prediction. Among the compared models, Hybrid Model-3 stood out, showcasing superior performance metrics and providing valuable insights into effective defect prediction strategies across projects.

Based on above, innovative PTP methodology for CI is proposed for utilizing the historical data for execution of test cases. The success and failure of new generated test cases is evaluated through machine learning classifiers like k-Nearest Neighbors, Random Forest, Support Vector Machine (SVM), Gradient Boosting, and Logistic Regression. ROC-AUC score, F1-score, recall, accuracy, and precision are computed for optimization and prioritization of the test cases for enhancing the functioning of the software testing efficiency. The computed results are depicted through tables and graphs.

III. METHODOLOGY

This section delves deeper into the systematic methodology that CI systems use to effectively use predictive modeling to select tests. The proposed methodology is predicated on the utilization of past test case execution data to inform more advanced techniques. By utilizing historical data analysis in predicting test case outcomes, the main goal is to enhance the testing procedure. In this regard, figure 1 illustrates the graphical representation of the proposed methodology for predicting the probability of test cases.

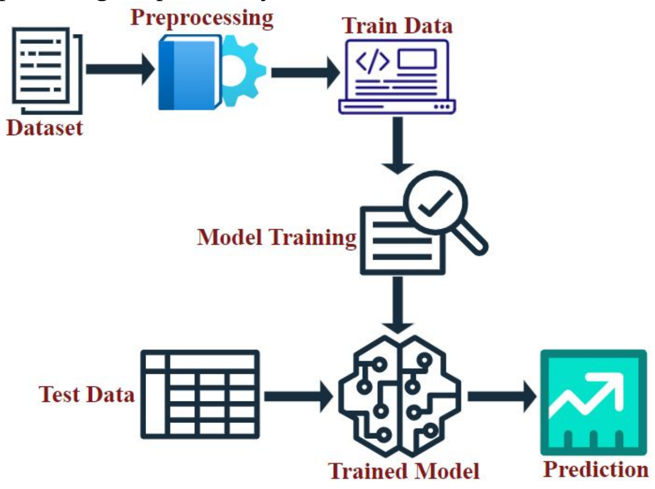
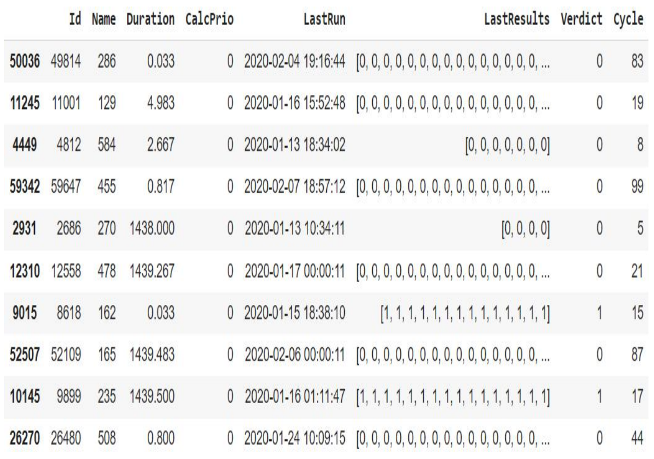


Figure 1: A System Model for PTP Approach

1. Dataset

The research utilizes a dataset representing the historical execution details of test cases within CI environment. This dataset spans a month and encompasses vital information such as test identifiers, test names, durations, prioritization scores, execution dates, previous outcomes, verdicts (pass or fail), and the corresponding CI cycle. The following figure 2 shows a visual representation of a representative sample of the dataset and gives a summary of the primary features and structure.



	Id	Name	Duration	CalcPrio	LastRun	LastResults	Verdict	Cycle
50036	49814	286	0.033	0	2020-02-04 19:16:44	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...	0	83
11245	11001	129	4.983	0	2020-01-16 15:52:48	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...	0	19
4449	4812	584	2.667	0	2020-01-13 18:34:02	[0, 0, 0, 0, 0, 0, 0]	0	8
59342	59647	455	0.817	0	2020-02-07 18:57:12	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...	0	99
2931	2686	270	1438.000	0	2020-01-13 10:34:11	[0, 0, 0, 0]	0	5
12310	12558	478	1439.267	0	2020-01-17 00:00:11	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...	0	21
9015	8618	162	0.033	0	2020-01-15 18:38:10	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]	1	15
52507	52109	165	1439.483	0	2020-02-06 00:00:11	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...	0	87
10145	9899	235	1439.500	0	2020-01-16 01:11:47	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]	1	17
26270	26480	508	0.800	0	2020-01-24 10:09:15	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...	0	44

Figure 2: A Sample of Dataset [19]

2. Objective

The present work aims to test case prioritization in CI systems by utilizing machine learning classifier models like Random Forest, Logistic Regression**, Support Vector Machine (**SVM), Gradient Boosting and K-Nearest Neighbors trained on historical test execution data, and predict the probability of failure for new test cases, thereby providing a data-driven and efficient approach to prioritize test cases. It revolutionizes testing strategies, offering a reliable framework to inform decision-making in software testing processes based on historical insights.

3. Feature Selection

To design the predictive model, two vital features have been identified:

- ***Duration:*** The time taken for a test case to execute;
- ***CalcPrio:*** The prioritization assigned to a test case in a CI cycle.

The selection of these features is predicated on how features might affect the results of test cases.

4. Model Training

At this crucial stage, various classifiers are trained to evaluate for prediction of of the outcomes of tests. Details about the selected classifiers and algorithms are provided as follows:

4.1. Random Forest

Forest is an ensemble learning technique that was first presented by Tin Kam Ho in 1995 [20]. It has shown promise in a variety of machine-learning applications. With its strong ensemble of decision trees, it does exceptionally well at identifying intricate correlations between datasets. The algorithm learns to respond to various patterns using test execution data from the past, generating a sophisticated foundation for predicting outcomes for new test cases. Because it is ensemble-based, it is robust and reduces overfitting, which enhances test prioritizing in CI environments. The algorithm is given below:

Random Forest Algorithm pseudo-code

```
Input:
  • Training dataset with features $X_{train}$ and labels $y_{train}$ .
  • Number of decision trees $n_{estimators}$, maximum depth ($max_{depth}$), and other hyperparameters.
Process:
  1. Construct a random forest with N random trees $H=\{h_1,h_2,\ldots,h_N\}$.
  2. Repeat until convergence:
       o For each random tree $h_i$ in H:
           ▪ Construct a concomitant ensemble $H_{-i}$.
           ▪ Use $H_{-i}$ to label all unlabeled data, estimate labeling confidence.
           ▪ Add data with confidence > $\theta$ to newly labeled set $L_t^i$.
           ▪ Undersample $L_t^i$ to satisfy a condition.
           ▪ Update hi by learning a random tree using $L \cup L_t^i$.
Output:
  • H, with predictions by majority voting from all trees.
```

4.2. Logistic Regression

A comprehensive history of logistic regression is provided by Cramer (2002) [21]. However, in the 1830's and 1840's, Pierre François Verhulst, working under Adolphe Quetelet, gave it the name "logistic" because of its first use as a population growth model. For binary and multiclass classification issues, logistic regression works incredibly well. It makes use of past data to identify correlations between input features and test case success or failure in the context of test prioritizing. The model contributes significantly to the predictive method because to its interpretability and simplicity. The algorithm is given below:

Logistic Regression Algorithm pseudo-code

```
Input:
  • Training dataset with features $X_{train}$ and labels $y_{train}$ .
  • Learning rate ($\alpha$), regularization term ($\lambda$), and other hyperparameters.
Process:
  1. Initialize weights W and bias b.
  2. Repeat until convergence:
       o Compute linear combination $z=X_{train}\cdot W+b$.
       o Apply sigmoid function $P(Y=1|X)=\frac{1}{1+e^{-z}}$.
       o Update weights using gradient descent.
Output:
  • Trained Logistic Regression model with weights W and bias b.
```

4.3. Support Vector Machine (SVM)

In 1995, Cortes and Vapnik at AT&T Bell Laboratories developed Support Vector Machines (SVM) [22]. SVM's adaptability to various data distributions and ability to manage complex decision boundaries account for its performance. When prioritizing test cases, SVM analyzes past test execution data to find patterns that influence a test case's success or failure. The algorithm is given below:

Support Vector Machine (SVM) Algorithm pseudo-code

```
Input:
  • Training dataset with features Xtrain and labels $y_{train}$.
  • SVM kernel, regularization term (C), and other hyperparameters.
Process:
  1. Using a kernel, transform input features into higher-dimensional space.
  2. Optimize:
       o Solve $\min_{w,b,\xi} 1/2\|w\|^2+C\sum_{i=1}^{N}\xi_i$ subject to constraints.
Output:
  • Trained SVM model with parameters w and b.
```

4.4. Gradient Boosting

Gradient boosting, an ensemble technique, uses several weak learners, most frequently decision trees, to produce a reliable and accurate model. The realization that boosting functions as an optimization technique on a cost function, as discovered by Leo Breiman [23], served as inspiration for it. Because of its ability to consistently correct errors from previous models, it is very skilled at seeing minute correlations and patterns in data. In the framework of continuous integration, test case prioritization establishes which test cases to appropriately prioritize based on historical execution data. The algorithm is given below:

Gradient Boosting Algorithm pseudo-code

```
Input:
  • Training dataset with features $X_{train}$ and labels $y_{train}$
  • Number of boosting stages (trees), learning rate ($\eta$), and other hyperparameters.
Process:
  1. Initialize predictions $F0(X)=mean(y_{train})$.
  2. Repeat for each boosting stage i:
       o Compute residuals $r_i=y_{train}-F_{i-1}(X)$.
       o Fit decision tree $T_i$ to residuals.
       o Update predictions $F_i(X)=F_{i-1}(X)+\eta\cdot T_i(X)$.
Output:
  • Trained Gradient Boosting model.
```

4.5. K-Nearest Neighbors (k-NN)

Evelyn Fix and Joseph Hodges developed the k-nearest neighbors algorithm (k-NN) in 1951[24]. It is a flexible non-parametric supervised learning technique for applications including regression and classification. K-NN uses the consensus of the k-nearest neighbors to assign labels or forecast values, based on the idea that neighboring points in feature space have similar properties. It locates comparable instances in feature space by utilizing the locality concept. Test case prioritizing is achieved via k-NN, which is particularly effective in scenarios with intricate and non-linear decision boundaries. It accomplishes this by employing historical test execution data for predicting the prioritization of new test cases. The algorithm is given below:

```
k-Nearest Neighbors (k-NN) Algorithm pseudo-code
Input:
  • Training dataset with features $X_{train}$ and labels $y_{train}$ .
  • Value of k (number of neighbors) and chosen distance metric.
Process:
  1. For each test point $X_{test}$:
       o Calculate distances to all training points.
       o Identify k-nearest neighbors.
       o Assign class label based on majority voting.
Output:
  • Trained k-NN model.
```

Each model gains a sophisticated understanding of the underlying patterns from past test data due to this structured training procedure. As a result, these models turn into predictive engines that provide an efficient data-driven paradigm for test prioritization in CI systems.

5. Model Evaluation

A comprehensive evaluation method using a specific testing set is applied to the trained the models to get an appropriate and comprehensive evaluation. Each model's performance is seen systemically via a suite of measures that are included in the evaluation as given below:

- **Accuracy:** The proportion of correct predictions out of total predictions.
- **Classification Report:** Provides precision, recall, and F1-score metrics for a comprehensive model assessment.
- **Confusion Matrix:** A detailed breakdown of model performance, distinguishing between true positives, true negatives, false positives, and false negatives.
- **ROC Curve and AUC Score:** The Receiver Operating Characteristic (ROC) curve visualizes the trade-off between sensitivity and specificity, with the Area Under the Curve (AUC) quantifying the model's ability to distinguish between classes.

6. Predictive Testing

Afterwards, new test cases with varied durations are predicted using the learned models. Each case's expected outcome and probability of failure are recorded.

7. Results Analysis

To find trends, advantages, and disadvantages that are specific to each model, the acquired findings are carefully analyzed. Understanding the model's performance is aided by visualizations such as confusion matrix. This investigation contributes to the understanding of each classifier's ability to predict.

The algorithm depicted below illustrates the sequential steps of a predictive modeling process using a variety of classifiers. It showcases the algorithmic stages involved in diverse classifiers, providing a comprehensive view of the precise methodology employed for generating predictions.

```
Classifier-Based PTP Algorithm
1. Load Dataset:
     ✓ dataset = pd.load_csv('dataset.csv')
2. Data Preprocessing:
     ✓ Features: features=dataset['Duration','CalcPrio']
     ✓ Target Variable: target=dataset['Verdict']
     ✓ Split into Training and Testing sets: $X_{train}$, $X_{test}$,$y_{train}$,$y_{test}$
       =train_test_split(features,target,test_size=0.2,random_state=42)
3. Loop Classifier:
     ✓ For each classifier classifier in ({RandomForest, LogisticRegression, SVM, GradientBoosting, kNN}:
          • Create a pipeline: model=make_pipeline(StandardScaler,classifier)
          • Train and Predict: model.fit($X_{train}$,$y_{train}$),$y_{pred}$=model.predict($X_{test}$)
4. Evaluation Loop:
     ✓ For each model:
          • Compute metrics:
               ➢Accuracy: accuracy=accuracy_score($y_{test}$,$y_{pred}$)
               ➢Classification Report: class_report=classification_report($y_{test}$,$y_{pred}$)
               ➢Confusion Matrix: conf_matrix=confusion_matrix($y_{test}$,$y_{pred}$)
5. Predictive Testing Loop:
     ✓ For each model:
```

- For a new_test_case:
  - ➢Predict Verdict: verdict_pred=model.predict(new_test_case)
  - ➢Predict Failure Probability: failure_prob=model.predict_proba(new_test_case)[:,1]

6. **Results Analysis:**
   - ✓ Analyze the overall performance metrics of classifiers, including the probability of failure or pass for each model.

The objective of such a broad strategy is to create, evaluate, and apply predictive models for test case prioritization. The ultimate objective is to improve the efficiency of software testing in environments that utilize continuous integration. In Continuous Integration (CI) systems, this methodology methodically enhances test prioritizing. It entails loading and preprocessing data, training and assessing classifiers iteratively, and predicting test case Pass and failure probabilities using a predictive testing loop. Informed decision-making in CI environments is helped by the results analysis step, which evaluates overall classifier performance.

IV. RESULTS AND DISCUSSION

In this section, a comprehensive analysis of results is derived from experimentation involving a diverse array of classifiers within the domain of test case prioritization within the CI framework. The principal objective is to systematically assess the performance of the models, delineate the inherent strengths and weaknesses, and investigate potential avenues for refining the predictive methodology. The experimentation entails the utilization of five distinct classifiers: Random Forest, Logistic Regression, Support Vector Machine (SVM), Gradient Boosting, and k-nearest Neighbors. Each classifier undergoes a rigorous training process on a dataset that encapsulates the historical records of test case executions within a Continuous Integration environment. Crucial performance metrics, encompassing accuracy, precision, recall, and the F1-score are methodically scrutinized using a specifically designated held-out test set for each classifier. Furthermore, the models are deployed to prognosticate the probabilities of success or failure for novel test cases. This approach offers valuable insights into the generalization capabilities of the models and provides an understanding of the confidence levels associated with each prediction. The evaluated parameters are discussed below in brief:

1. Accuracy

Accuracy measures the overall correctness of a classification model. It represents the ratio of correctly predicted instances to the total number of instances. The accuracy is obtained by

$$Accuracy = \frac{True\ Positives + True\ Negatives}{Total\ Instances} \quad (1)$$

In the above, True Positives represent instances correctly predicted as positive (typically denoting the minority class, e.g., "Fail"), True Negatives denote instances correctly predicted as negative (usually representing the majority class, e.g., "Pass"), and Total Instances signify the entire dataset. Table 1 presents a comparative analysis of various classifiers, showcasing their respective accuracy scores. Additionally, figure 3 visually depicts the accuracy performance of each classifier, providing a graphical representation for ease of interpretation. This evaluation methodology enables a nuanced understanding of how well each classifier performs across different scenarios and highlights the distinctions in predictive capabilities.

Table 1: Classifier Performance Comparison with Accuracy Scores

Classifier	Accuracy
Random Forest	87%
Logistic Regression	79%
SVM	79%
Gradient Boosting	83%
k-Nearest Neighbors	86%

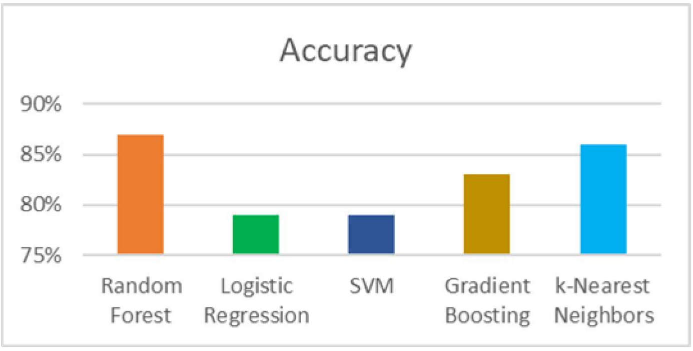


Figure 3: Accuracy Comparison of Different Classifiers

The Random Forest classifier excelled with an 87% accuracy, adeptly discerning between Pass and Fail instances in historical test data. Logistic Regression and SVM showed comparable accuracies of 79%, addressing linear and non-linear scenarios. Gradient Boosting achieved 83% accuracy, highlighting ensemble learning efficacy. Notably, k-Nearest Neighbors reached 86% accuracy, capturing intricate patterns. These metrics inform predicting new test cases, aiding strategic classifier selection for nuanced test prioritization. Identified strengths offer a robust foundation for precise real-world decisions in prioritizing tests efficiently.

2. Precision (Class 0 and Class 1)

Precision assesses the accuracy of positive predictions made by the model. It is particularly relevant in scenarios where the cost of false positives is high. A high precision indicates that when the model predicts a positive instance, it is likely correct. It is computed by

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives}$$

Where, True Positives denote instances accurately predicted as positive, while False Positives signify instances erroneously predicted as positive.

Table 2 and Figure 4 furnish an exhaustive examination of precision scores across diverse classifiers, discerning between Class 0 and Class 1 predictions. Each precision value is computed based on the corresponding true positives and false positives for the respective class, thereby providing nuanced insights into the models' proficiency in making precise positive predictions.

Table 2: Precision Comparison for Class 0 and Class 1

Classifier	Precision (Class 0)	Precision (Class 1)
Random Forest	0.89	0.76
Logistic Regression	0.79	0.00
SVM	0.79	0.00
Gradient Boosting	0.82	0.92
k-Nearest Neighbors	0.90	0.69

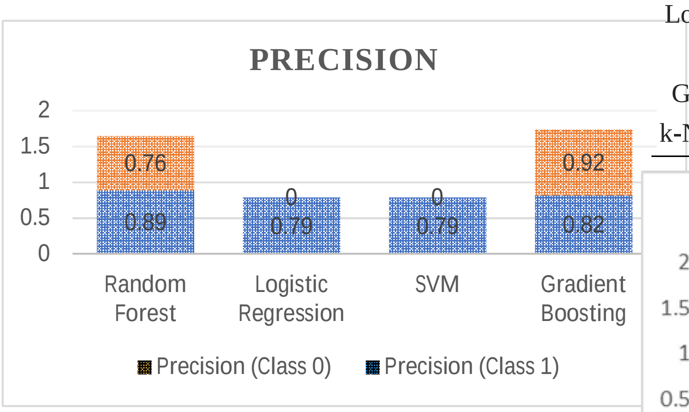


Figure 4: Precision Comparison for Class 0 and Class 1

Precision scores elucidate the predictive prowess of each classifier in discerning Pass (Class 0) and Fail (Class 1) instances within historical test data. Specifically, the Random Forest classifier attains a precision of 0.89 for Pass and 0.76 for Fail, signifying robust predictive capabilities. Logistic Regression demonstrates 0.79 precision for Pass but registers 0.00 for Fail, a performance mirrored by SVM whereas, Gradient Boosting performs outstandingly, as evidenced by precision evaluations of 0.92 for Fail and 0.82 for Pass. The k-Nearest Neighbors approach has a precision of 0.69 for a fail and 0.90 for a pass. These precision metrics assume paramount importance in predicting the failure or pass probabilities of new test cases, offering strategic insights for selecting models tailored to efficient test prioritization.

3. Recall (Class 0 and Class 1):

Recall measures the ability of a model to capture all positive instances. It is crucial in situations where missing positive instances is a significant concern. A high recall indicates that the model effectively identifies most of the positive instances. It is computed by

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives} \quad (3)$$

Where, True Positives denote instances accurately predicted as positive, while False Negatives represent instances incorrectly predicted as negative.

Table 3 provides a comprehensive comparative analysis of recall scores among different classifiers, distinguishing between Class 0 and Class 1 predictions. The recall values shed light on the models' effectiveness in accurately capturing positive instances, offering valuable insights into their overall performance. Additionally, figure 5 graphically illustrates their respective recall performances.

Table 3: Classifier Recall Comparison for Class 0 and Class 1

Classifier	Recall (Class 0)	Recall (Class 1)
Random Forest	0.95	0.55
Logistic Regression	1.00	0.00
SVM	1.00	0.00
Gradient Boosting	1.00	0.19
k-Nearest Neighbors	0.93	0.60

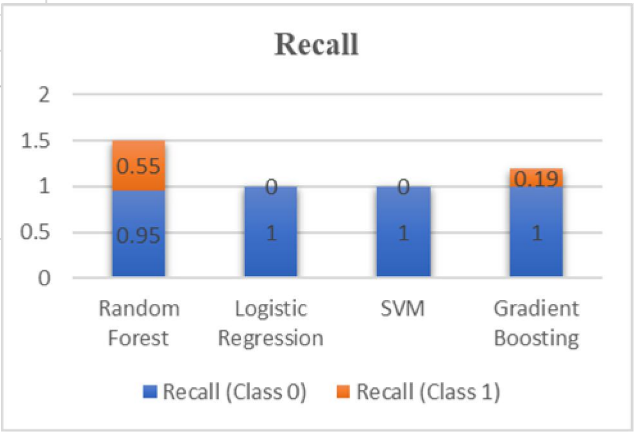


Figure 5: Graphical Representation of Classifier Recall Performance

The presented table outlines recall scores for each classifier, revealing their effectiveness in capturing instances within Class 0 (Pass) and Class 1 (Fail) based on historical test data. Random Forest demonstrates robust recall (0.95) for Pass but a relatively lower recall (0.55) for Fail. Logistic Regression achieves perfect Pass recall but zero Fail recall, mirrored by SVM. Gradient Boosting achieves perfect Pass recall but exhibits limited Fail recall (0.19). Conversely, k-Nearest Neighbors achieves high Pass recall (0.93) and moderate Fail recall (0.60). These recall values provide nuanced insights into each classifier's sensitivity to correctly identifying positive instances within each class based on historical data.

4. F1-Score (Class 0 and Class 1)

F1-score is the harmonic mean of precision and recall. It provides a balance between precision and recall and is especially useful when there is an uneven class distribution. It is computed by

$$F1-Score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \qquad (4)$$

Precision and Recall are computed for Class 0 and Class 1. Table 4 provides a detailed comparison of F1-Scores across various classifiers, delineating the performance concerning Class 0 and Class 1 predictions. The F1-Score values encapsulate the models' proficiency in achieving a harmonious balance between precision and recall, offering insights into their effectiveness across diverse class distributions. Figure 6 graphically portrays this balance.

Table 4: F1-Score Comparison for Class 0 and Class 1

Classifier	F1-Score (Class 0)	F1-Score (Class 1)
Random Forest	0.92	0.63
Logistic Regression	0.88	0.00
SVM	0.88	0.00
Gradient Boosting	0.90	0.32
k-Nearest Neighbors	0.91	0.64

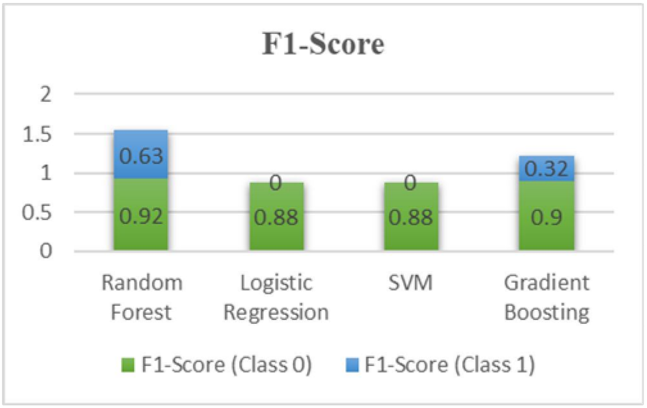


Figure 6: Graphical Representation of F1-Score Balance

Table 4 provides detailed F1-Score metrics for each classifier, offering a nuanced evaluation of precision and recall performance for Class 0 (Pass) and Class 1 (Fail) based on historical test data. Random Forest achieves a robust F1-Score of 0.92 for Pass instances, demonstrating a harmonious balance between precision and recall, while the F1-Score of 0.63 for Fail instances reflects a commendable compromise. Logistic Regression exhibits a commendable F1-Score of 0.88 for Pass instances, showcasing a balanced synthesis of precision and recall. However, the absence of F1-Score for Fail instances (marked as 0.00) underscores challenges in achieving a balanced performance for actual Fail instances. SVM mirrors Logistic Regression's performance with a respectable F1-Score of 0.88 for Pass instances but lacks any F1-Score for Fail instances, highlighting the struggle to achieve a balanced performance for actual Fail instances Gradient Boosting demonstrates a high F1-Score of 0.90 for Pass instances, indicating a balanced precision-recall trade-off. However, the F1-Score of 0.32 for Fail instances suggests challenges in achieving a balanced performance for actual Fail instances. k-Nearest Neighbors exhibits a high F1-Score of 0.91 for Pass instances, signifying a balanced performance in correctly identifying actual Pass instances. The F1-Score of 0.64 for Fail instances reflects a balanced precision-recall trade-off for actual Fail instances. These F1-Score metrics offer a granular assessment of each classifier's nuanced performance, essential for predicting the probability of failure or pass for new test cases based on historical data.

5. Confusion Matrix

A confusion matrix is a table that provides a detailed breakdown of the model's predictions. It shows the counts of true positives, true negatives, false positives, and false negatives. The elements of the matrix represent different outcomes based on the actual and predicted class labels.

$$\begin{bmatrix} True\ Negatives & False\ Positives \\ False\ Negatives & True\ Positives \end{bmatrix}$$

The confusion matrix for the Random Forest model illustrates the following outcomes: True Negatives (TN) - 9630, False Positives (FP) - 477, False Negatives (FN) - 1222, True Positives (TP) - 1476. This indicates that the model accurately identified 9630 instances as "Pass" (class 0) and 1476 instances as "Fail" (class 1). However, it mistakenly classified 477 instances as "Fail" when they were "Pass" and 1222 instances as "Pass" when they were "Fail."

The confusion matrix can be visually represented using a heatmap, providing a clearer illustration in the figure 7.

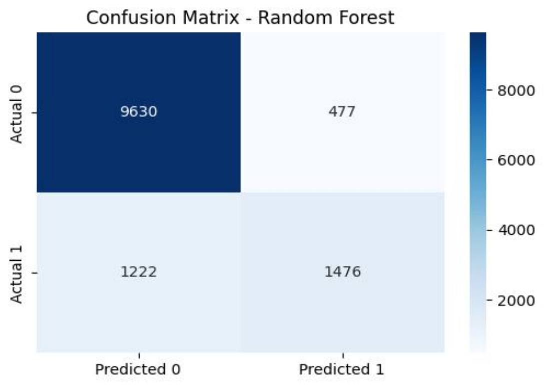


Figure 7: Confusion Matrix of Random Forest

In both Logistic Regression and Support Vector Machines (SVM), the confusion matrix reveals uniform values: True Negatives (TN) amounting to 10,107, False Positives (FP) registering at 0, False Negatives (FN) totaling 2,698, and True Positives (TP) standing at 0. In the context of

Gradient Boosting, the confusion matrix displays TN: 10,063, FP: 44, FN: 2,175, and TP: 523. For the k-Nearest Neighbors (k-NN) algorithm, the confusion matrix records TN: 9,375, FP: 732, FN: 1,076, and TP: 1,622. Figure 8 visually illustrates the performance metrics in a heatmap encompassing all the models.

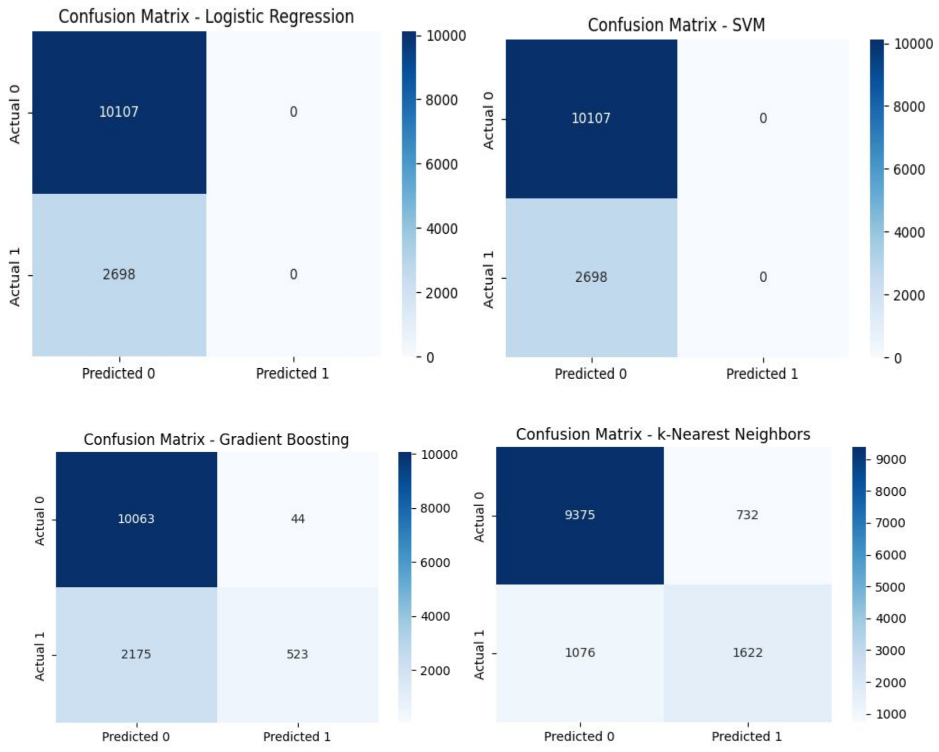


Figure 8: Metrics Heatmap for Logistic Regression, SVM, Gradient Boosting, and k-NN Models

V. Conclusions

This present study explores optimizing test prioritization in CI environments using machine learning classifiers. The Random Forest classifier outperformed others with an 87% accuracy, excelling in distinguishing pass and fail outcomes. Logistic Regression and SVM faced challenges with an imbalanced dataset. Including confusion matrices, provided nuanced insights. For instance, the Random Forest model exhibited 9630 true negatives, 477 false positives, 1222 false negatives, and 1476 true positives. While contributing to intelligent test prioritization in CI, future work should focus on refining models, exploring additional features, and addressing dataset imbalances for enhanced CI pipeline efficiency. The predictive approach showcases automation potential, optimizing resource utilization and elevating software quality. As software development evolves, leveraging machine learning in test prioritization signifies a progressive shift. Ongoing research can further refine models and adapt approaches to diverse CI scenarios, shaping the future of software testing.

REFERENCES

[1] Durelli, V. H., Durelli, R. S., Borges, S. S., Endo, A. T., Eler, M. M., Dias, D. R., & Guimarães, M. P. (2019). Machine learning applied to software testing: A systematic mapping study. *IEEE Transactions on Reliability*, *68*(3), 1189-1212, DOI: https://doi.org/10.1109/TR.2019.2892517.

[2] Speiser, J. L., Miller, M. E., Tooze, J., & Ip, E. (2019). A comparison of random forest variable selection methods for classification prediction modeling. *Expert systems with applications*, *134*, 93-101, DOI: https://doi.org/10.1016/j.eswa.2019.05.028.

[3] Zhu, C., Idemudia, C. U., & Feng, W. (2019). Improved logistic regression model for diabetes prediction by integrating PCA and K-means techniques. *Informatics in Medicine Unlocked*, *17*, 100179, DOI: https://doi.org/10.1016/j.imu.2019.100179.

[4] Marijan, D., & Gotlieb, A. (2020, April). Software testing for machine learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34, No. 09, pp. 13576-13582, DOI: https://doi.org/10.1609/aaai.v34i09.7084.

[5] Yucalar, F., Ozcift, A., Borandag, E., & Kilinc, D. (2020). Multiple-classifiers in software quality engineering: Combining predictors to improve software fault prediction ability. *Engineering Science and Technology, an International Journal*, *23*(4), 938-950, DOI: https://doi.org/10.1016/j.jestch.2019.10.005.

[6] Meçe, E. K., Paci, H., & Binjaku, K. (2020). The application of machine learning in test case prioritization-a review. *European Journal of Electrical Engineering and Computer Science*, *4*(1), DOI: https://doi.org/10.24018/ejece.2020.4.1.128.

[7] Braiek, H. B., & Khomh, F. (2020). On testing machine learning programs. *Journal of Systems and*

Software, *164*, 110542, DOI: https://doi.org/10.1016/j.jss.2020.110542.

[8] Khatibsyarbini, M., Isa, M. A., Jawawi, D. N., Shafie, M. L. M., Wan-Kadir, W. M. N., Hamed, H. N. A., & Suffian, M. D. M. (2021). Trend application of machine learning in test case prioritization: A review on techniques. *IEEE Access*, *9*, 166262-166282.

[9] Goyal, J., & Ranjan Sinha, R. (2022). Software defect-based prediction using logistic regression: Review and challenges. In *Second International Conference on Sustainable Technologies for Computational Intelligence: Proceedings of ICTSCI 2021* (pp. 233-248). Springer Singapore, ISBN: 978-981-16-4640-9.

[10] Gezici, B., & Tarhan, A. K. (2022, September). Explainable AI for Software Defect Prediction with Gradient Boosting Classifier. In *2022 7th International Conference on Computer Science and Engineering (UBMK)* (pp. 1-6). IEEE, DOI: https://doi.org/10.1109/UBMK55850.2022.9919490.

[11] Omri, S. (2022). *Quality-Aware Learning to Prioritize Test Cases* (Doctoral dissertation, Karlsruhe Institute of Technology, Germany), DOI: http://dx.doi.org/10.5445/IR/1000143079.

[12] Pan, R., Bagherzadeh, M., Ghaleb, T. A., & Briand, L. (2022). Test case selection and prioritization using machine learning: a systematic literature review. *Empirical Software Engineering*, *27*(2), 29, DOI: https://doi.org/10.1007/s10664-021-10066-6.

[13] Yaraghi, A. S., Bagherzadeh, M., Kahani, N., & Briand, L. C. (2022). Scalable and accurate test case prioritization in continuous integration contexts. *IEEE Transactions on Software Engineering*, *49*(4), 1615-1639, DOI: https://doi.org/10.1109/TSE.2022.3184842.

[14] Da Roza, E. A., Lima, J. A. P., Silva, R. C., & Vergilio, S. R. (2022, March). Machine learning regression techniques for test case prioritization in continuous integration environment. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)* (pp. 196-206). IEEE, DOI: https://doi.org/10.1109/SANER53432.2022.00034.

[15] Marijan, D. (2023). Comparative study of machine learning test case prioritization for continuous integration testing. *Software Quality Journal*, 1-24, DOI: https://doi.org/10.1007/s11219-023-09646-0.

[16] Sánchez-García, A. J., López-Martín, C., & Abran, A. (2023). Gradient Boosting Optimized Through Differential Evolution for Predicting the Testing Effort of Software Projects. *IEEE Access*, *11*, 135235-135254, DOI: https://doi.org/10.1109/ACCESS.2023.3337809.

[17] Ramesh, L., Radhika, S., & Jothi, S. (2023). Hybrid support vector machine and K-nearest neighbor-based software testing for educational assistant. *Concurrency and Computation: Practice and Experience*, *35*(1), e7433, DOI: https://doi.org/10.1002/cpe.7433.

[18] Kumar, H., & Saxena, V. (2024). Software Defect Prediction Using Hybrid Machine Learning Techniques: A Comparative Study. *Journal of Software Engineering and Applications*, *17*(4), 155-171, DOI: https://doi.org/10.4236/jsea.2024.174009.

[19] https://www.kaggle.com/datasets/joolousada/test-case-prioritization-data (Accessed on 20 Dec. 2023).

[20] Ho, T. K. (1995, August). Random decision forests. In *Proceedings of 3rd international conference on document analysis and recognition* (Vol. 1, pp. 278-282). IEEE, DOI: https://doi.org/10.1109/ICDAR.1995.598994.

[21] Cramer, J.S., The Origins of Logistic Regression (December 2002). *Tinbergen Institute Working* Paper No. 2002-119/4, DOI: http://dx.doi.org/10.2139/ssrn.360300.

[22] Cortes, C., & Vapnik, V. (1995). Support-vector networks. *Machine learning*, *20*, 273-297, DOI: https://doi.org/10.1007/BF00994018.

[23] Breiman, L. (1997). *Arcing the edge* (pp. 1-14). Technical Report 486, Statistics Department, University of California at Berkeley, Available on https://statistics.berkeley.edu/sites/default/files/tech-reports/486.pdf.

[24] Fix, E., & Hodges Jr, J. L. (1951). Discriminatory analysis: Nonparametric discrimination: Consistency properties.

[25] Saxena, D. V., & Raj, D. (2010). Local Area Network Performance Using UML. *International Journal of Advanced Networking and Applications*, *2*(02), 614-620.