A Data-Driven Approach Towards Software Regression Testing Quality Optimization

Abdallah Moubayed*, Nouh Alhindawi†‡, Jamal Alsakran§, MohammadNoor Injadat¶, Mohammad Kanan‖
*Computer Engineering Department and Interdisciplinary Research Center for Intelligent Secure Systems,
King Fahd University of Petroleum & Minerals, Dhahran, Saudi Arabia; e-mail: abdallah.moubayed@kfupm.edu.sa
†School of Computing and Augmented Intelligence, Arizona State University, Gilbert, Arizona, USA; e-mail: nalhinda@asu.edu
‡Department of Software Engineering, Jadara University, Irbid, Jordan; e-mail: hindawi@jadara.edu.jo
§Fairleigh Dickinson University, Canada; e-mail: j.alsakran@fdu.edu
¶Data Science & Artificial Intelligence Department, Zarqa University, Zarqa, Jordan; e-mail: minjadat@zu.edu.jo
‖ Department of Industrial Engineering, University of Business and Technology, Jeddah, Saudi Arabia; e-mail: m.kanan@ubt.edu.sa

2024 25th International Arab Conference on Information Technology (ACIT) | 979-8-3315-4001-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/ACIT62805.2024.10877022

***Abstract*—Software testing is very important in software development to ensure its quality and reliability. As software systems have become more complex, the number of test cases has increased, which presents the challenge of executing all the tests in a limited time frame. Various test case prioritization techniques have been introduced to solve this problem. These methods aim to identify and implement the most critical tests first. In this paper, we propose an implementation of a dynamic test case prioritization strategy to improve software quality by increasing code coverage with special attention to edge case handling. Edge case test prioritization is a technique that improves test efficiency by selecting extreme case scenarios that can reveal critical bugs or unexpected behavior early in development, improving overall software reliability and dependability. In order to prioritize test cases, this paper presents a regression-based method that makes use of machine learning algorithms. The approach leverages previous performance data to optimize regression testing efficiency by examining variables like test time and execution status. Performance evaluations, when compared against industry standards and cutting-edge techniques, show how effective these algorithms are at correctly prioritizing test cases and identifying faults. This study offers simplified yet reliable solutions for regression testing optimization by shedding light on the efficacy of regression algorithms, such as Random Forest and decision trees.**

***Index Terms*—Software Testing Optimization, Machine Learning, Natural Language Processing**

I. Introduction

Software testing is a crucial component of the software development life-cycle. One example technique is regression testing. Regression testing is the practice of re-running functional and non-functional tests on an application to verify that, following updates, enhancements, or modifications, it continues to operate and perform as intended [1]. This approach is essential to the software development life-cycle because it facilitates the identification of errors or defects that may result from unanticipated changes to the code base [2]. By rerunning the tests, developers can ensure that the program continues functioning as intended and is free of errors or regressions resulting from product modifications. In the absence of a version control system, it can be difficult to find the component responsible for the error. Regression testing, on the other hand, explains the origin of the problem, which facilitates more efficient troubleshooting. It performs routine maintenance of the software and provides an overview of its status.

Due to its repetitive nature, regression testing lends itself well to automation, making it an ideal choice for automated testing. In the context of regression testing, edge cases pertain to the analysis of situations that exist at the outermost or boundary points of input ranges, conditions, or requirements. These scenarios are atypical in nature and are yet critical to test, as they may unveil errors or vulnerabilities that were introduced during code modifications. By adding edge cases, testers try to confirm the robustness and reliability of the software in several different scenarios and proactively identify regressions or unintended consequences caused by changes. Examples include testing with minimum and maximum input values, evaluating compatibility with older versions, and evaluating performance under stressful conditions. This approach provides comprehensive coverage and helps uncover hidden problems early in development, improving overall software quality and reliability [3][4].

Through creativity and critical thinking, QA testers identify extreme test cases and bugs and actively look for scenarios that are at the extremes or limits of normal usage during any test cycle, be it regression, functionality, or at any stage of the development process. By understanding the customer's workflow and exploring known integration points, testers can find common ground. They also consider how users can deviate from the expected path and intentionally perform unexpected actions [5].

The prioritization of software testing is critical for efficient resource allocation. However, usually not every scenario can be prioritized. Instead, we focus on critical areas and features that significantly affect the functionality, reliability, and user experience of the software. High-risk scenarios, commonly used features, and core features are prioritized based on factors such as severity, frequency, impact on user experience, business criticality, and likelihood of exploitation [6]. Edge case scenarios that represent extreme or unusual usage scenarios are also prioritized in the same way, with an emphasis on mitigating risk, maintaining user satisfaction, and patching potential vulnerabilities. Although each case cannot be individually prioritized, efforts are directed to those with the greatest possible impact within the boundaries of the project.

Integrating machine learning (ML) into prioritization within software testing is extremely important and constantly evolving [7,8]. ML algorithms can analyze large data sets, including historical test results, bug logs, code changes, and user input to independently prioritize test cases, defects, or features. These prioritization levels are determined based on several factors, including severity, impact, frequency, and business importance. Patterns and trends in the data are absorbed by the ML models seamlessly to identify high-priority items better and more efficiently than traditional manual

methods. In addition, ML-based prioritization is adaptive and improves over time as it continuously learns from additional data and feedback, making it an important tool for improving testing activities and resource utilization. Overall, ML significantly improves software test prioritization processes by facilitating more efficient and targeted test strategies.

ML testing outperforms automated testing with its superior data analysis, customizability, and scalability capabilities. ML testing independently prioritizes test cases and defects by analyzing complex data patterns, resulting in more accurate defect detection. Unlike automated testing, ML testing continuously learns from new data, which incrementally improves test performance. Its adaptability allows it to effectively manage different testing needs and evolving software environments. Overall, ML testing provides a more advanced and comprehensive approach to software testing, ensuring the delivery of higher-quality software products.

To that end, in this research, various ML algorithms are used such as Support Vector Machines (SVM), random forest, k-nearest neighbors (kNN), and neural networks (specifically multilayer perceptrons or MLP) to improve the prioritization process. These techniques have proven effective for a multitude of applications including natural language processing and spam filtering [9]–[14], automatic speech recognition [15], computer vision [16,17], educational data mining [18], and network security [19]–[22].

The paper structure is as follows: Section II summarizes some of the previous literature on this topic. Section III describes the data collection and data processing along with a brief overview of the ML models considered as well as the evaluation metrics used. Section IV outlines and discusses the results achieved. Finally, Section V summarizes the work and provides multiple potential research avenues worth exploring.

II. Related Works

Prioritizing test cases is an important step in software testing, particularly for complex systems. There are various ways of prioritizing test cases based on characteristics including code coverage, risk, and fault-proneness. These strategies have disadvantages, such as subjectivity and lack of comprehensiveness. Machine learning techniques may enhance the efficiency and effectiveness of test case prioritizing. Machine learning algorithms may find patterns in data and use them to anticipate outcomes. Several researches have examined the use of machine learning techniques to prioritize test cases. In recent years, machine learning algorithms have gained traction in the field of test case prioritization due to their ability to analyze large datasets and extract meaningful patterns.

Lachmann *et al.* [23] compared Support Vector Machine (SVM) to standard test case prioritization methods such as coverage, text path, text content, history, age, and random ordering. The data showed that SVM consistently outperforms other evaluation measures, particularly average recall and average percentage of fault detected rating. SVM has noticeable benefits in that it requires much fewer top-ranked tests to attain the necessary recall average, as well as higher APFD scores, indicating improved default detection capacity. Conversely, the standalone use of code coverage appears less influential, albeit still integral to SVM's overall effectiveness. These results underscore SVM's potential to revolutionize test case prioritization methodologies, offering valuable insights for optimizing software testing processes in diverse industrial contexts.

Busjaegr *et al.* [24] suggested a test case prioritization strategy based on SVM in which historical data is used to train a binary classifier for ordering test cases. This strategy outperforms non-ML based alternatives in terms of defect detection. SVM can effectively rank test cases based on their ability to detect flaws resulting in more efficient testing operations. Similarly, Lity *et al.* [25] used SVM-Rank and SVM-based ranking algorithms to prioritize test cases based on failure rates. The examination revealed that SVM-based techniques beat expert-led manual prioritization methods. Using this test case can be prioritized in a way that best targets fault detection efforts, increasing overall testing efficiency and effectiveness.

Grano *et al.* [26] extended this concept by developing a regression predictive model for assessing test branch coverage that included SVM and Random Forest. By anticipating branch coverage, this approach makes it easier to generate test cases for continuous integration testing. The experimental results demonstrated SVM's robust fault prediction accuracy, particularly when prioritizing test cases.

In the realm of predictive test prioritization, recent studies have explored the efficacy of boosted decision tree approaches, to optimize the testing process. Machalica's method [27], uses a boosted decision tree to train a classifier that can predict the likelihood of a test case failing based on code modification and a selection of test cases. This strategy has shown exceptional effectiveness, reducing testing expenses by a factor of two while retaining a high individual test failure detection rate of more than 95%.

Similarly, Chen *et al.* [28] suggested another XGBoost-based predictive test prioritizing strategy. This method entails doing a test case distribution analysis to assess the fault detection capability of actual regression testing. Chen's method has been empirically confirmed and deployed in real-world settings, resulting in a significant reduction in testing expense.

Ahemad A. Saifan [29] proposed a method for reducing test cases generated automatically using KNN and a clustering algorithm based on cyclomatic complexity and code coverage. Using these parameters, the strategy attempted to efficiently minimize the number of test cases while maintaining test efficacy. However, misunderstanding emerges when two modules have similar cyclomatic complexity and code coverage. This issue has been widely documented in past studies when modules with the same complexity and coverage metrics might exhibit different behavior or functionality. As a result, relying entirely on these indicators to reduce test cases may lead to crucial test scenarios with the possibility of misidentifying modules with equivalent complexity and coverage.

Two machine learning in continuous integration were examined in the study by Miranda *et al.* [30] along with an assessment of several ML algorithms. Using supervised learning, the first technique, Learning to Rank, trained the model on test features and then ranks test sets in the following commits. However, because this approach may become outdated as context changes, retraining of the model is required. It is discovered that Ranking to Learn is more suited for CI setting since it emphasizes Reinforcement Learning. More resilience is shown by the RL-based approach in terms of test case volatility, code modifications, and quantity of failed tests.

The RETECS [31] technique uses Reinforcement learning

algorithms to present an innovative way of prioritizing test cases. In order to efficiently prioritize and choose test cases, RETECS takes into account test case duration, historical failure data, and prior execution cycles. The main characteristic of RETECS is that it uses a reward function that adjusts over time to accommodate system modifications. Similarly, Multi-Armed Bandit algorithms are used by COLEMAN [32], an alternate reward function method. These methods strike a compromise between diversity and the number of new test cases, addressing volatility and the Every Error (EvE) problem. Despite their effectiveness, both RETECS and COLEMAN possess certain limitations. They do not take into account the current state of the system for prioritization, and their performance diminishes when dealing with large test case sets.

III. Methodology & Implementation

In this study, we propose the use of different ML algorithms to prioritize test cases and evaluate their effectiveness compared to traditional prioritization methods. The methodology used in this study is described below.

A. Data Collection:

Our study involved gathering data from open source GitHub projects, which included attributes for the project such as duration of the feature, priority, date of the last run and feature changes. The dataset used here is from Cisco [33] comprises 356 entries with 13 features. Here's a summary of its key aspects:

- Name: This field is an identifier for test cases or products, with a wide range of values.
- Duration: Indicates execution time of the test case, with a significant variation across entries.
- LastRun: A date field indicating when the last run occurred. The dates vary across a range, suggesting a timeline for the dataset's events.
- E1, E2, E3: Binary indicators (0 or 1) that represent execution status of the particular test case in the last 3 sprints.
- Verdict: Another binary indicator, signifying the outcome of a test or evaluation as pass (1) or fail (0).
- DIST: An integer that represents a measure of distance or difference, related to changes in test conditions or versions.
- CHANGE_IN_STATUS: Indicates changes, with values ranging from 0 to 2, tracking status transitions.
- PRIORITY_VALUE: A floating-point number that could denote the importance or urgency of attending to the respective entry, with a broad range of values.
- Cycle: This denotes a cycle number or iteration for each test entry, showing that entries undergo different multiple test cycles .
- LastRunFeature and DurationFeature: Floating-point numbers, derived features from LastRun and Duration respectively, for analytical purposes.

The dataset includes a mix of identifiers, binary indicators, integer and floating-point numbers, and a date field. This structure suggests a focus on tracking and analyzing test executions, outcomes, and their attributes over time, fitting well with a research topic aimed at optimizing software quality through dynamic test case prioritization. The features such as duration, verdict, and priority value are directly relevant, offering a foundation for investigating how test prioritization can impact software testing efficiency and effectiveness, particularly in identifying and focusing on critical edge cases.

B. Data Processing and Model Selection:

The next step is to process the data so that the machine could understand it and use ML algorithms to make predictions based on the processed data. In this study, four different ML models are selected such as Support Vector Machine (SVM), k-Nearest Neighbors (kNN), Decision Trees, and Random Forest to predict the priority values for test cases in a Cisco dataset based on various features. These models are selected based on their strengths and suitability for software test prioritization.

1) Support Vector Machine: SVM is a flexible algorithm that can handle both simple and complex data patterns. It is widely known for handling high-dimensional data where we have more features than data points. The accuracy of predictions in limited data is achieved by focusing on most important data points for making separations [34]. SVM handles different data patterns by using kernel methods by transforming data features with the help of kernel functions. These kernel functions make data separation easier by mapping complex datasets to higher dimensions with the help of hyperplanes. This hyperplane acts as a decision boundary [34].

As can be seen in Fig. 1, anything that falls to one side of the decision boundary, will be classified as blue, and anything that falls to the other as red [34]. However, there can be different hyperplanes, where the best hyperplane can be defined as the one whose distance to the nearest element of each tag is the largest [34]. In Fig. 2, the text "large margin" refers to the space between the solid lines, which is considered the optimal separation. The text "small margin" refers to the space between the dashed lines, which is a less desirable separation [34].

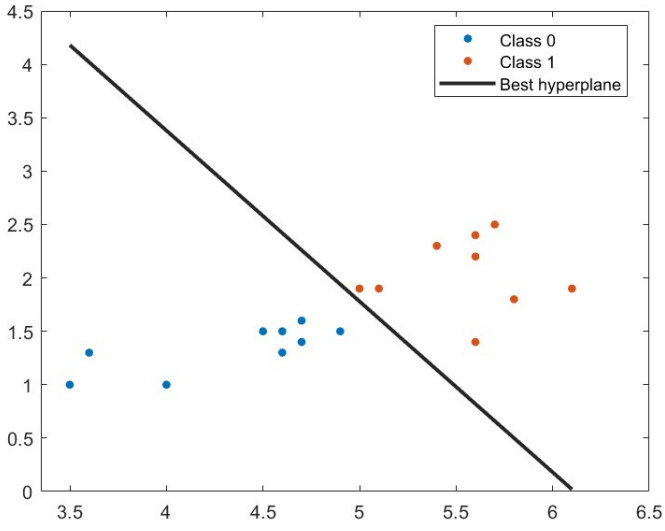

Fig. 1. Two-dimensional Best Hyperplane

In this study, an SVM model is trained on the data, makes predictions, and compares its performance to other models. This demonstrates how SVMs can be integrated into real-world tasks and their effectiveness in handling complex datasets.

2) K-Nearest Neighbor (kNN): kNN is a simple ML algorithm which can be used for both classification and regression tasks.

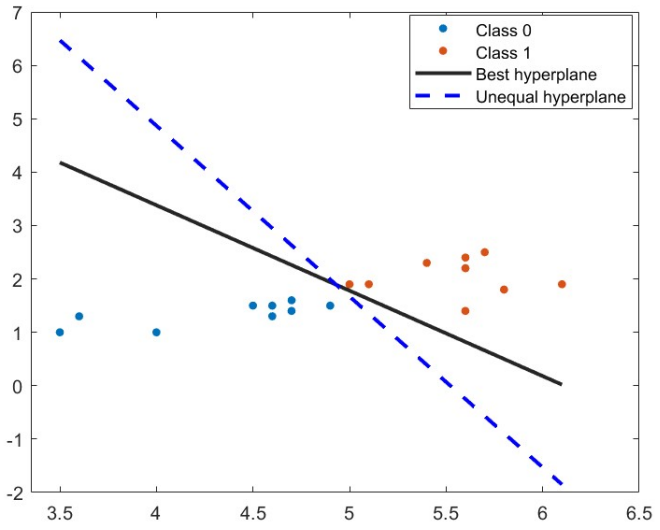

Fig. 2. Two-dimensional Unequal Hyperplane

kNN simply assumes that similar things can be found in close proximity [35]. In regression, new value of data is predicted based on the nearest neighbor in the training data. This process involves choosing a value of K, the number of nearest neighbors, that will be used to make predictions. This is a hyperparameter that you can tune based on the characteristics of your data. A small K (e.g., 1 or 3) may lead to noisy predictions, while a large K may lead to overly smoothed predictions. While making predictions, kNN selects K different data points closest to the current data point by calculating the distance between this point and the other data points. The average of the target values of the K nearest neighbors is the anticipated value for the new data point in a regression. This may be a straightforward arithmetic mean [35].

The implemented kNN model in this study will select the five closest data points for the predictions. The model uses the features as input and target variables i.e. priority of the test cases as values to to predicted. In the next step, the trained kNN model is used to make predictions on the testing data by identifying the five closest neighbors from the training data and predicts the target value by averaging the target values of those neighbors.

3) Decision Trees: Decision trees are a type of supervised machine learning algorithm that classify or regress data through binary answers to specific questions. The structure resembles a tree, consisting of different node types: root, internal, and leaf nodes. The root node serves as the starting point, splitting into internal and leaf nodes. Leaf nodes represent the final classification categories or actual values [36].

To create a decision tree, a function is first defined to serve as the root node. Typically, no single feature can perfectly predict final outcomes, a situation referred to as an attachment. Techniques like Gini impurity, entropy, and data acceleration are employed to assess how well a function classifies the data. At each level, the feature with the least impurity is chosen as the node. To compute the Gini coefficient for a numerical feature, the data is sorted in ascending order, and the means of adjacent values are averaged. The Gini coefficient is then determined for each selected mean, ranking data points based on whether their feature values are below or above this mean and assessing how accurately this choice classifies the data. The Gini coefficient is calculated using the formula provided (Eq. 1), where K is the number of rating categories and p is the proportion of occurrences in those categories [36].

$$\textit{Gini Impurity} = 1 - \sum_{i=1}^{K} p_i^2 \tag{1}$$

For each value, the weighted average of Gini impurity scores for leaves is calculated. The value with the fewest impurities is selected for this property. The process is repeated for different attributes to select the attribute and value that will become the node. This process is repeated at each node at each depth level until all data have been classified. Once the tree is built, the model can make a prediction from a data point by moving down the tree, using the conditions at each node to get the final value. When using decision trees for regression, the sum of squares of the residuals or variance is used instead of the Gini to measure additive [36].

In this study, Decision trees divide the feature space into regions, giving each region a constant value that is the average of the target values. The decision tree technique divides the dataset into subsets recursively depending on features during training in order to minimize a selected criterion (such as mean squared error). A fresh sample moves through the tree depending on feature values, moving from the root node to a leaf node in order to make predictions. As for the sample, the constant value related to the leaf node is the prediction. In this instance, a mapping between the input characteristics and the priority values is learned by the decision tree regressor. Decision trees implicitly rank features according to how important they are to the prediction of the target variable, in this case, priority values.

4) Random Forest: Random Forests are an ensemble learning ML algorithm based on decision trees. An ensemble learning technique is used for regression in the supervised learning algorithm known as random forest regression. In order for Random Forest to function, a large number of decision trees are built during training, and the classes that result are the mean prediction (regression) or the mode of the classes (classification) of each individual tree [37].

A random forest aggregates numerous decision trees with a few useful tweaks, making it a meta-estimator (i.e., combining the outcome of multiple predictions):

- The ability of each node to be split into a number of features is limited to some percentage of the total (which is known as the hyper-parameter). This restriction makes sure that all features that have the potential to be predictive are fairly used by the ensemble model and that no feature is overly dependent on any one another [37].
- When generating splits, each tree draws a random sample from the original data. adding an extra layer of randomization and preventing overfitting [37].

In this study, Random Forest is used for regression to rank test cases according to projected priority values. The model is assessed using testing data that hasn't been seen before and trained using training data. The number of decision trees to be created is specified to be 100 in this case. Independently, every decision tree in the forest picks up patterns from a randomized selection of features and data points.

In the next step, the trained model makes predictions on the testing data. The ultimate prediction of the forest is the average

of the predictions made by each decision tree individually. Test case prioritization and additional analysis are made possible by joining the anticipated priority values with the original dataset. Furthermore, the dataset is sorted based on the predicted priority values, prioritizing test cases with higher predicted importance.

C. Evaluation Metrics:

While ML models are useful tools, they cannot be used reliably until their results are carefully evaluated. Model evaluation is the term for this evaluation procedure. It includes evaluating a model's performance using hypothetical data and aids in troubleshooting. Users can get more confident in the models' forecasts and steer clear of making important judgments based on erroneous outcomes by examining them [38].

This study includes evaluation of following metrics:

- Mean Absolute Squared Error(MAE): The average absolute difference between the target values and the forecasted values is determined by the Mean Absolute Error (MAE) measure. Better performance is indicated by a lower MAE, with a value of 0 denoting flawless prediction. This metric is not applicable to classification models [38].
- Root Mean Squared Error (RMSE): RMSE calculates the average magnitude of errors between values that are predicted and those that are observed. It is comparable to MAE. However, it highlights bigger errors by squaring them before performing the average operation, giving bigger errors more weight. Better performance is shown by a lower RMSE, with a value of 0 denoting flawless prediction [38].
- R-squared (R^2): This measure shows how closely the model's regression line matches the actual data points. It shows the percentage of the target variable's variance that the characteristics of the model can account for. A better fit is indicated by a value that is closer to 1, while a perfect match is indicated by a value of 1. It is crucial to remember that R-squared might be deceptive in certain situations, especially when working with a large number of characteristics. R-squared might not be relevant if a model is not a regression model or if it does not generate continuous predictions [38].
- Mean Absolute Percentage Error (MAPE): The error is expressed as a proportion of the actual target values using the MAPE statistic. It facilitates error comparison across several data sets. Better performance is indicated by a lower MAPE; a value of 0 denotes perfect prediction. MAPE might not be important for tasks involving discrete classification [38].

IV. Experimental Results & Discussion

A. Results' Discussion:

The evaluation's findings showed that models produced the best results for all of the metrics used. This shows that the underlying relationships in the data were successfully captured by these models, and they produced precise predictions for the target variable.

Based on the evaluation metrics, both Random Forest and Decision Tree stand out as best options for the prioritization of test cases. These two models have average forecasts that are the closest to the target values, as indicated by their lowest Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE) values. They obtain extremely high R-squared values that are very near to 1, which indicates that the observed data and the model's predictions fit each other well. Additionally, their Mean Absolute Percentage Error (MAPE), which indicates how accurate their forecasts are in relation to the target variable values' magnitudes, is the lowest or very competitive.

TABLE I
ML Models' Performance Evaluation

Metric	SVM	kNN	Decision Tree	Random Forest
MAE	0.057	0.013	0.006	0.007
RMSE	0.066	0.067	0.023	0.032
R^2	0.9822	0.9814	0.9978	0.9957
MAPE	135.26%	4.07%	1.41%	1.38%

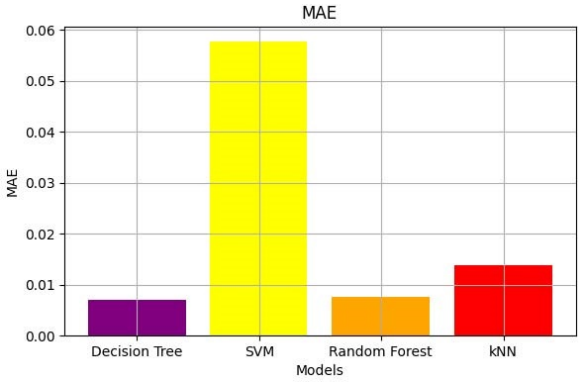

Fig. 3. Mean Absolute Error (MAE)

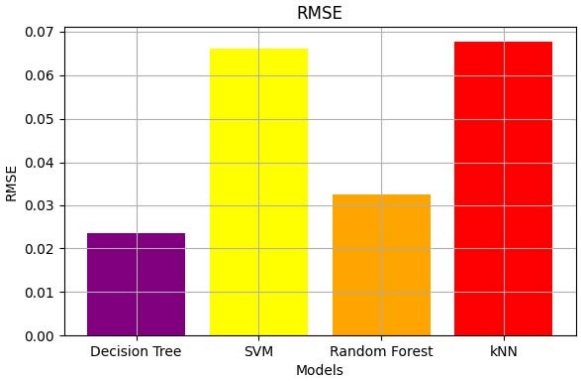

Fig. 4. Root Mean Squared Error (RMSE)

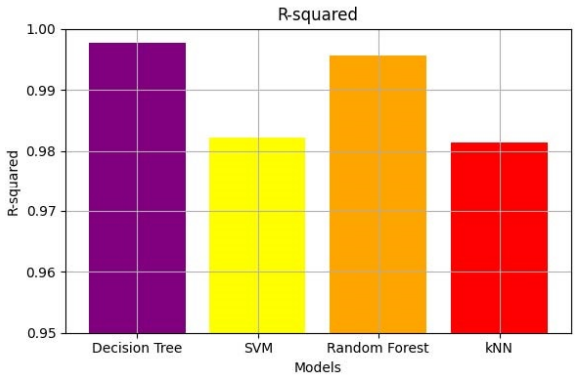

Fig. 5. R-squared Values

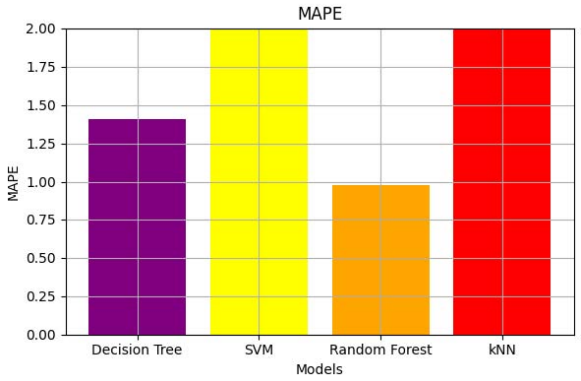

Fig. 6. Mean Absolute Percentage Error

Random Forest is thus chosen for the next stages of test case prioritization since it typically performs better in predicting priority values than other models like SVM, Decision Tree, or kNN in this scenario. After training the Random Forest model and predicting the priority values for existing edge test cases, new edge test cases are created based on their predicted priority values. This strategy makes use of the strong predictive powers of the Random Forest model to guarantee that test cases are carried out in a way that optimizes the possibility of identifying possible problems early in the testing process. Furthermore, the efficacy of regression testing may be further improved by contributing to more effective and efficient testing processes through the creation of edge test cases and ranking them alongside current test cases.

To create edge cases for regression testing, an approach should be followed to identify extreme scenarios at the input domain's limits or extremities. These scenarios could be identified by detailed examination of the software's needs and specifications. Boundary criteria are then established for every input parameter, including the lowest and highest allowable value and values that fall between and above these boundaries.

In this research, attributes for the edge cases are defined in a similar fashion to the dataset used, along with verdicts to these edge cases assigned randomly either as "pass" or "fail". By generating diverse sample edge cases, the random forest model is can be tested for test case prioritization.

B. Work Limitations:

It is worth noting that the current work suffers from some limitations. The first limitation is the scope of the data. The Cisco dataset [33] is considered to relatively small (356 entries) and may not fully represent larger, diverse projects. This is attributed to the lack of publicly available datasets related to complex software projects since many top companies refrain from sharing this type of information in an open access manner. The second limitation of this work is related to the code coverage. While the study focuses on prioritizing edge cases, overall code coverage gains were limited. This suggests potential refinement of prioritization strategies. Hence, there is a need to investigate the impact of those prioritization strategies in improving the code coverage metric.

V. Conclusion & Future Work

This study implemented and evaluated different models like SVM, decision trees, random forests, and kNN for software quality optimization through dynamic test case prioritization. Experimental results showed that Tree-based models such as Random Forest and Decision Tree stand out as the best options for the prioritization of test cases. These two models had average forecasts that are the closest to the target values, as indicated by their lowest Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE) values. Additionally, they achieved extremely high R-squared values that are very near to 1, indicating that the observed data and the model's predictions fit each other well. Moreover, their Mean Absolute Percentage Error (MAPE) is the lowest or very competitive.

While the findings of this study show the potential of employing ML models for dynamic test case prioritization with a focus on edge cases, there are various areas for future research and development.

One of the research's primary goals was to maximize code coverage through prioritized test case selection with particular emphasis on edge cases. However, efforts to significantly boost code coverage with existing methodologies were unsuccessful. Despite applying models such as neural networks, a substantial improvement in overall code coverage measure was not observed. There are several possible explanations for the lack of improvement. First, the benchmark dataset used may already have quite thorough test suites, providing less possibility for coverage gains based only on better prioritization. Furthermore, prioritizing edge situations may reduce overall coverage in exchange for greater fault identification in corner cases.

Moving forward, multi-objective optimization with edge case priority and overall coverage maximization could be investigated. Treating the two as independent may produce better outcomes than attempting to maximize coverage as a proxy. Innovating new coverage measures based on edge case scenarios may also provide better targets to optimize against.

The dataset used in this study, while real-world data source from industry had some limitations. It contained a relatively small number of test cases (356) and features (13) compared to the potential scale of production software system with thousands of test cases. To increase the applicability and reliability of the ML models, acquiring and incorporating larger, more diverse datasets would be highly valuable. Data from additional companies projects of varying scale and domains, and wider test case feature sets could enable training higher capacity models. Techniques like transfer learning, where models are pre trained on abundant data then fine tuned on spare targets, may also unlock performance gains. Expanding the dataset would reduce overfitting risks and enable the model to learn more generalizable patterns for effective prioritization.

Additionally, there is a vast opportunity to explore more advanced and specialized model architectures. Architecture tailored for sequence prediction, like recurrent neural networks, transformers, and other deep learning models, could potentially extract richer patterns from the temporal aspect of historical test case verdicts and prioritize more accurately. Customized attention mechanisms could emphasize important factors like recent failures.

Ensemble methods combining different types of models in an optimal way yield more robust, higher performance prioritization compared to individual model types. Automated model architecture search could also discover novel architectures better suited to unique characteristics of the test prioritization task.

The models evaluated in this study used an offline, batch-learning approach where historical data was used to train a fixed model

which then made prioritization predictions on new test cases. However, an online learning paradigm could potentially improve performance. In an online or reinforcement learning setup, the prioritization model could continuously update and adapt itself based on the ground truth outcomes of its prioritization decisions on new test cases as they run. This closes the loop between model outputs and newly observed data. For example, if an unexpectedly severe bug slipped through testing on a low-priority test case, that case could be up-weighted in importance automatically for future prioritization. Exploration vs exploitation techniques could balance validating current beliefs with probing potentially higher-value areas. This type of online learning system could respond faster to changing testing priorities, environments, and immediate feedback compared to the static models used initially. It could overcome the limitations of making prioritization decisions based solely on historical data snapshots which may become stale or not fully generalizable.

While the research focused specifically on prioritizing test cases for regression testing of software changes, the general techniques explored could translate to many other applications of intelligent prioritization and scheduling new feature development based on estimated business impact, customer importance, dependencies, etc. Intelligent labeling and prioritization of defects, bug reports, and security issues based on severity and urgency.

REFERENCES

[1] M. Qasim, A. Bibi, S. J. Hussain, N. Jhanjhi, M. Humayun, and N. U. Sama, "Test case prioritization techniques in software regression testing: An overview," *International Journal of advanced and applied sciences*, vol. 8, no. 5, pp. 107–121, 2021.

[2] H. Alsawalqah, H. Faris, I. Aljarah, L. Alnemer, and N. Alhindawi, "Hybrid smote-ensemble approach for software defect prediction," in *Software Engineering Trends and Techniques in Intelligent Systems*, R. Silhavy, P. Silhavy, Z. Prokopova, R. Senkerik, and Z. Kominkova Oplatkova, Eds. Cham: Springer International Publishing, 2017, pp. 355–366.

[3] A. Zarrad, "A systematic review on regression testing for web-based applications." *J. Softw.*, vol. 10, no. 8, pp. 971–990, 2015.

[4] H. Do, "Recent advances in regression testing techniques," *Advances in computers*, vol. 103, pp. 53–77, 2016.

[5] D. Hao, L. Zhang, L. Zang, Y. Wang, X. Wu, and T. Xie, "To be optimal or not in test-case prioritization," *IEEE Transactions on Software Engineering*, vol. 42, no. 5, pp. 490–505, 2015.

[6] U. Badhera, G. Purohit, and D. Biswas, "Test case prioritization algorithm based upon modified code coverage in regression testing," *International Journal of Software Engineering & Applications*, vol. 3, no. 6, p. 29, 2012.

[7] N. Taba and S. Ow, "A new model for software inspection at the requirements analysis and design phases of software development," *The International Arab Journal of Information Technology*, vol. 13, no. 6, pp. 644–651, 2016.

[8] O. Meqdadi, N. Alhindawi, J. Alsakran, A. Saifan, and H. Migdadi, "Mining software repositories for adaptive change commits using machine learning techniques," *Information and Software Technology*, vol. 109, pp. 80–91, 2019.

[9] E. Aljadani, F. Assiri, and A. Alshutayri, "Detecting spam reviews in arabic by deep learning," *The International Arab Journal of Information Technology*, vol. 21, no. 3, pp. 495–505, 2024.

[10] A. Alyabrodi, S. Al-Daja, M. Injadat, A. Moubayed, and H. Khattab, "Enhancing the performance of sign language recognition models using machine learning," in *2023 24th International Arab Conference on Information Technology (ACIT)*, 2023, pp. 01–07.

[11] S. Alzu'bi, O. Badarneh, B. Hawashin, M. Al-Ayyoub, N. Alhindawi, and Y. Jararweh, "Multi-label emotion classification for arabic tweets," in *2019 Sixth International Conference on Social Networks Analysis, Management and Security (SNAMS)*, 2019, pp. 499–504.

[12] F. Abdullah, M. Al-Ayyoub, I. Hmeidi, and N. Alhindaw, "A deep learning approach to classify and quantify the multiple emotions of arabic tweets," in *2021 12th International Conference on Information and Communication Systems (ICICS)*. IEEE, 2021, pp. 399–404.

[13] H. Gharaibeh, R. E. Al Mamlook, G. Samara, A. Nasayreh, S. Smadi, K. M. Nahar, M. Aljaidi, E. Al-Daoud, M. Gharaibeh, and L. Abualigah, "Arabic sentiment analysis of monkeypox using deep neural network and optimized hyperparameters of machine learning algorithms," *Social Network Analysis and Mining*, vol. 14, no. 1, p. 30, 2024.

[14] E. Al-Daoud, G. Samara, M. R. A. Sara, S. Taqatqa, and M. Kanan, "Exploring the effectiveness of different embedding methods for toxicity classification," in *Artificial Intelligence and Economic Sustainability in the Era of Industrial Revolution 5.0*. Springer, 2024, pp. 233–241.

[15] K. M. Nahar, F. Al-Omari, N. Alhindawi, and M. Banikhalaf, "Sounds recognition in the battlefield using convolutional neural network," *International Journal of Computing and Digital Systems*, vol. 11, no. 1, pp. 189–198, 2022.

[16] A. B. Moghlebey, H. Haydar, A. Moubayed, and M. Injadat, "Impact of application choice on itracker model performance evaluation: A perspective," in *2023 2nd International Engineering Conference on Electrical, Energy, and Artificial Intelligence (EICEEAI)*, 2023, pp. 1–6.

[17] E. Al Daoud and G. Samara, "Improving the face recognition performance using gabor and vggface2 features concatenation," in *2022 6th International Conference on Information Technology (InCIT)*. IEEE, 2022, pp. 187–190.

[18] A. Moubayed, M. Injadat, N. Alhindawi, G. Samara, S. Abuasal, and R. Alazaidah, "A deep learning approach towards student performance prediction in online courses: Challenges based on a global perspective," in *2023 24th International Arab Conference on Information Technology (ACIT)*, 2023, pp. 01–06.

[19] A. Moubayed, M. Injadat, T. M. Abdellatif Mohamed, S. Almatarneh, M. Al-Mashagbeh, and M. Aljaidi, "Securing smart homes using hybrid stacking ensemble deep learning-enabled framework," in *2023 2nd International Engineering Conference on Electrical, Energy, and Artificial Intelligence (EICEEAI)*, 2023, pp. 1–6.

[20] M. Salman, B. bani slman, M. Aljaidi, R. b. Saleem, A. Alsarhan, M. H. Qasem, M. N. Injadat, and B. Igried, "A study of forensic tools data recovery performance," in *2023 2nd International Engineering Conference on Electrical, Energy, and Artificial Intelligence (EICEEAI)*, 2023, pp. 1–6.

[21] R. Alazaidah, A. Al-Shaikh, M. Al-Mousa, H. Khafajah, G. Samara, M. Alzyoud, N. Al-Shanableh, and S. Almatarneh, "Website phishing detection using machine learning techniques," *Journal of Statistics Applications & Probability*, vol. 13, no. 1, pp. 119–129, 2024.

[22] A. Alnatsheh, A. Alsarhan, M. Aljaidi, H. Rafiq, K. Mansour, G. Samara, B. Igried, and Y. A. Al Gumaei, "Machine learning-based approach for detecting ddos attack in sdn," in *2023 2nd International Engineering Conference on Electrical, Energy, and Artificial Intelligence (EICEEAI)*, 2023, pp. 1–5.

[23] R. Lachmann, S. Schulze, M. Nieke, C. Seidl, and I. Schaefer, "System-level test case prioritization using machine learning," in *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2016, pp. 361–368.

[24] B. Busjaeger and T. Xie, "Learning for test prioritization: an industrial case study," in *Proceedings of the 2016 24th ACM SIGSOFT International symposium on foundations of software engineering*, 2016, pp. 975–980.

[25] S. Lity, R. Lachmann, M. Lochau, and I. Schaefer, "Delta-oriented software product line test models-the body comfort system case study," *Technical report, TU Braunschweig*, 2013.

[26] G. Grano, T. V. Titov, S. Panichella, and H. C. Gall, "How high will it be? using machine learning models to predict branch coverage in automated testing," in *2018 IEEE workshop on machine learning techniques for software quality evaluation (MaLTeSQuE)*. IEEE, 2018, pp. 19–24.

[27] M. Machalica, A. Samylkin, M. Porth, and S. Chandra, "Predictive test selection," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 91–100.

[28] 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.

[29] A. A. Saifan *et al.*, "Test case reduction using data mining classifier techniques." *J. Softw.*, vol. 11, no. 7, pp. 656–663, 2016.

[30] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino, "Fast approaches to scalable similarity-based test case prioritization," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 222–232.

[31] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement learning for automatic test case prioritization and selection in continuous integration," in *Proceedings of the 26th ACM SIGSOFT international symposium on software testing and analysis*, 2017, pp. 12–22.

[32] J. A. P. Lima, W. D. Mendonça, S. R. Vergilio, and W. K. Assunção, "Learning-based prioritization of test cases in continuous integration of highly-configurable software," in *Proceedings of the 24th ACM conference on systems and software product line: Volume A-Volume A*, 2020, pp. 1–11.

[33] M. Bagherzadeh, N. Kahani, and L. Briand, "Reinforcement learning for test case prioritization," *IEEE Transactions on Software Engineering*, vol. 48, no. 8, pp. 2836–2856, 2021.

[34] B. Stecanella, "Support vector machines (svm) algorithm explained," *MonkeyLearn*, 2017. [Online]. Available: https://monkeylearn.com/blog/introduction-to-support-vector-machines-svm/

[35] K. Taunk, S. De, and S. Verma, "Machine learning classification with k-nearest neighbors," in *2019 International Conference on Intelligent Computing and Control Systems (ICCS), DOI: 0.1109/ICCS45141*, 2019.

[36] O. C. Njoku, "Decision trees and their application for classification and regression problems," 2019.

[37] A. Chakure and B. Whitfield, "Random forest regression in python explained," 2023. [Online]. Available: https://builtin.com/data-science/random-forest-python

[38] V. Plevris, G. Solorzano, N. P. Bakas, and M. E. A. Ben Seghier, "Investigation of performance metrics in regression analysis and machine learning-based prediction models," in *8th European Congress on Computational Methods in Applied Sciences and Engineering (ECCOMAS Congress 2022)*. European Community on Computational Methods in Applied Sciences, 2022.