KMANPUB | Decision Science and Intelligent Systems Vol. 3 No. 1 (2026): 1-15

Article history:
Received 13 January 2026
Revised 20 February 2026
Accepted 25 February 2026
Published online 01 March 2026

Machine Learning-Driven Test Case Prioritization Through Data Analytics in Software Testing

Eisa. Naji[1], Amir. Najafi[2*], Amir. Najafi[3]

[1] PhD Candidate, Department of Industrial Management, Qa.C., Islamic Azad University, Qazvin, Iran
[2] Associate Professor, Department of Industrial Engineering, NT.C., Islamic Azad University, Tehran, Iran
[1] Associate Professor, Department of Industrial Management, Qa.C., Islamic Azad University, Qazvin, Iran

*** Corresponding author email address**: asdnjf@gmail.com

A r t i c l e I n f o

Article type:
Original Research

How to cite this article:
Naji, E., Najafi, A., & Ehtesham Rasi, R. (2026). Machine Learning-Driven Test Case Prioritization Through Data Analytics in Software Testing. *Decision Science and Intelligent Systems. 3(1), 1-15.*

A B S T R A C T

Software testing plays a critical role in maintaining the dependability and overall quality of software applications. Nevertheless, it often demands significant time and computational resources, especially in unit testing environments. With the continuous expansion of test suites, running every available test case is no longer a practical strategy. Test Case Prioritization (TCP) addresses this challenge by arranging test cases in an order that increases the likelihood of detecting faults earlier while minimizing testing effort and cost. Conventional TCP techniques are often limited in their ability to adapt to the fast-paced nature of current software development practices. To overcome these shortcomings, this research explores the application of machine learning (ML) methods within the TCP framework to improve prioritization effectiveness. In contrast to fixed or rule-based techniques, ML-based approaches can learn patterns from historical test data and make adaptive prioritization decisions. For this purpose, a dataset was generated using several metaheuristic methods, including Genetic Algorithm (GA), Particle Swarm Optimization (PSO), Grey Wolf Optimizer (GWO), and a hybrid GWO model, implemented on the triangle classification benchmark problem. The collected test cases were categorized into five classes according to coverage criteria and then prepared for model training. Four supervised ML classifiers were developed and assessed using stratified 10-fold cross-validation. Their performance was evaluated based on widely used metrics, including accuracy, precision, recall, F1-score, and AUC. Experimental findings showed that the proposed approach reached an accuracy of 97.31% and an F1-score of 92.94%, surpassing traditional TCP methods. These findings indicate that integrating ML into TCP can improve early fault discovery, decrease unnecessary test executions, and offer an automated and scalable approach for more effective unit testing.

Keywords: *Software Testing, Test Case Prioritization, Machine Learning Techniques, Unit Test Optimization, Metaheuristic Methods.*

Introduction

Software testing remains one of the most critical and resource-intensive phases of the software development life cycle (SDLC), as it ensures that applications function correctly, efficiently, and securely (Lachmann, 2018). Software testing represents the process of evaluating and verifying that a software application satisfies required specifications and performs as intended (Pandhare, 2025). By detecting defects and missing requirements, testing reinforces quality, reliability, and performance (Khambam et al., 2022). Effective testing enhances user satisfaction and mitigates the risk of costly failures in deployed systems (Muhammad, 2024). It can be performed using several alternative methods and techniques, encompassing various activities throughout the software development process (Konidena et al., 2021). Standard testing methods include unit testing, integration testing, system testing, acceptance testing, and performance and regression testing.(Lonetti, 2018). Of various types of software testing, unit testing constitutes a particularly indispensable activity. It emphasizes validation that recent code modifications do not introduce unintended faults into previously verified functionality. However, as software systems scale in complexity and size, the associated testing costs have escalated proportionally (Ramachandran, 2026). As highlighted by Sharma (2024), unit testing may consume up to 80% of the overall testing effort, primarily due to the repeated execution of large test suites after every code change. Test case prioritization (TCP), a significant challenge in software testing, seeks to establish the optimal ordering of test case execution, thereby facilitating earlier fault detection and maximizing benefits to the tester by saving time and resources. It enables testers to identify critical test cases first, reducing the time and cost of testing (Ramzan et al., 2026). Test case prioritization emphasizes high-coverage tests, ensuring broader validation of the software under test. Executing programs with prioritized test data facilitates the early detection and correction of defects, thereby promoting a more efficient development process (Sivaraman, 2020). It optimizes the time, resources, and costs associated with software testing. Traditional TCP approaches, however, depend extensively on structural program data, such as code coverage statistics or past execution profiles. In most industrial environments, such information is not readily available or economical to obtain(Mukherjee & Patnaik, 2021). Additionally, most traditional TCP methods are computationally costly, making them infeasible for large-scale or real-time environments.

Despite advancements in test case prioritization (TCP) techniques, significant challenges persist in achieving scalability, adaptability, and automation in software testing. Traditional TCP methods often rely on detailed structural data, such as source code metrics, control flow analysis, or execution history, which can be time-consuming and costly to collect in large-scale systems. Khan et al. (2023) noted that traditional prioritization strategies often struggle with scalability and can result in limited accuracy when applied to diverse and complex test suites. These limitations render traditional TCP impractical for dynamic, large-scale projects that require rapid and efficient testing. According to Sharma et al. (2024), machine learning-based classification techniques can enhance the effectiveness of test prioritization by reducing execution time, improving the precision of fault detection, and highlighting the need for intelligent, automated alternatives, such as ML-based methods. Therefore, this thesis proposes an approach that addresses the shortcomings of traditional methods by integrating metaheuristic algorithms for test data generation with supervised ML models for prioritization. The goal of this study is to develop an automated, scalable TCP framework that reduces unit testing time, lowers costs, and enhances fault detection accuracy in large-scale software systems.

Test case prioritization (TCP) has emerged as a critical technique in software engineering to enhance the efficiency of unit testing processes. As software systems evolve and expand, the number of test cases grows exponentially. Modern applications can have thousands of test cases, making exhaustive testing impractical(Marijan, 2023; Sakhrawi, 2024). This growth makes it impossible to execute all tests within reasonable time and budget constraints. The fundamental objective of TCP is to order test cases in a manner that maximizes early fault detection, thereby providing faster feedback to development teams and reducing overall testing costs.(Wang, 2023). As Sakhrawi et al. (2024) note, software testing consumes a significant portion of development resources, with some estimates suggesting that testing can account for up to 50% of total development costs. By prioritizing test cases that are more likely to reveal faults, organizations can identify critical issues sooner, reducing the cost and complexity of fixes.(Sugave, 2025). Modern software development is further complicated by frequent updates and modifications. Traditional prioritization techniques, which rely heavily on historical execution data, face limitations in such dynamic environments, where test cases may be added, removed, or redefined depending on shifting project requirements (Bertolino, 2020).

These limitations have driven the investigation of machine learning (ML) and artificial intelligence (AI) in software testing. ML-based TCP approaches employ classification models to prioritize test cases within a suite, optimizing objectives such as early fault detection, maximizing code coverage, or minimizing testing time. These approaches are particularly effective at identifying faults early in the testing cycle, thereby increasing productivity and reducing overall testing costs. Recent empirical studies, such as the work by Son et al. (2025) on large-scale systems like SAP HANA, have demonstrated the superior performance of ML-based prioritization compared to traditional methods. Moreover, these models offer a scalable and adaptive solution that reduces manual effort while enhancing both speed and accuracy.

Methods and Materials

This section outlines the initial phase of the methodology, as shown in Figure 2, focusing on preparing benchmark programs and analyzing their source code. This step is critical, as the selected programs serve as the foundation for generating test data and prioritizing test cases in the subsequent step. Software defects, or bugs, can be categorized into two main types: syntax errors and logic errors. Syntax errors break programming language rules, such as missing semicolons or incorrect keywords, stopping the program from compiling. Logic errors, however, occur when syntactically correct code produces wrong or unexpected results due to poor design or reasoning. These errors are harder to detect because the program may function correctly for some inputs but fail for others.

To evaluate the proposed framework, this study uses the Triangle classification problem as a benchmark. The analysis focuses on the source code of a method with the following metrics:

- Inputs: The method takes three integer inputs (a, b, c) representing the side lengths of a triangle.
- Output: It returns an integer indicating the triangle.

Static analysis of the source code reveals the logic for classifying triangles:

- Lines 1-8: Define the C# namespace, the Triangle class, the method signature, and an integer variable (train) for classification.
- Lines 9-10: Validate inputs by checking if any side length is zero or negative, returning 0 for invalid triangles.

- Lines 11-17: Encode side equality into the train variable using complex logic to identify equal sides.
- Line 18: Check if the train equals 0, indicating a scalene triangle (no equal sides), and return 1.
- Lines 19-21: Perform the triangle inequality test, ensuring the sum of any two sides exceeds the third. If the test fails, return 0 (invalid); otherwise, confirm that the triangle is scalene and return 1.
- Lines 23-24: Check if the train exceeds 3 (e.g., train equals 6), identifying an equilateral triangle and returning 3.
- Lines 25-30: Handle isosceles triangle cases by checking the train and the triangle inequality, returning 2 for valid cases.
- Line 31: Return 0 as a catch-all for isosceles cases failing the triangle inequality (e.g., inputs 1, 1, 4).

This analysis identifies unexecuted code lines that may hide logic errors, posing risks of undetected bugs. The subsequent step of the methodology generates test data using metaheuristic algorithms and prioritizes test cases with machine learning to target these untested regions, enhancing code coverage and fault detection.

After preparing the benchmark programs, we proceed to the second phase of our automated test data generation. The goal is to generate a comprehensive set of test inputs that maximizes code coverage, particularly targeting unexecuted code regions that may hide logic errors.

The test data generation approach in this study builds on the Traxtor method proposed by Arasteh and Hosseini (2022). Their work, "Traxtor: An Automatic Software Test Suite Generation Method Inspired by Imperialist Competitive Optimization Algorithms," utilizes the Imperialist Competitive Algorithm (ICA) to generate test data that maximizes branch coverage efficiently. Traxtor achieved 99.99% average coverage, a 99.94% success rate, and an average generation time of 0.12 seconds, outperforming other methods. This approach, illustrated in Figure 3-3, serves as a blueprint for our method and includes three steps:

- Preprocessing: Static analysis extracts structural properties, such as conditional branch expressions, from the program's source code, aligning with the Triangle program analysis in Section 3.2.
- Generation: A metaheuristic algorithm (ICA) searches for test data guided by a fitness function that evaluates how well inputs cover untested branches, with a focus on hard-to-reach code.
- Evaluation**:** The generated test data is assessed for coverage, efficiency (in terms of time and iterations), and stability (Arasteh, 2022).

This study adopts Traxtor's structure and extends it by implementing and comparing five test data generation strategies to create diverse input sets for machine learning-based prioritization

To generate real-world test data, five types of metaheuristic algorithms are used: Random, GA, PSO, GWO, and Hybrid GWO. These test data are designed to maximize code coverage and fault detection.

1. Random Generation: This simple strategy generates test data by randomly selecting integer values within a defined range (e.g., -1000 to 1000). It serves as a baseline for comparison due to its lack of guided intelligence. While easy to implement, random generation is inefficient, often failing to cover edge cases or complex logic.

2. Genetic Algorithm (GA**)**: GA, an evolutionary algorithm, begins with a population of randomly generated test data. Each test case is evaluated for code coverage using a fitness function. High-performing test cases are selected as parents, producing new test cases through crossover (combining parts of two test cases) and mutation (randomly altering values). This process iteratively improves coverage, targeting untested code paths more effectively than random generation.
3. Particle Swarm Optimization (PSO): PSO, a swarm intelligence algorithm, models test data as particles in a search space. Each particle's position (input values) and velocity are adjusted based on its best-known position (pbest) and the swarm's best position (gbest). This balances exploration and exploitation, efficiently generating test data that covers complex code paths.
4. Grey Wolf Optimizer (GWO): GWO, another swarm intelligence algorithm, mimics the hunting behavior of grey wolves. Test data is organized into a hierarchy with the alpha (best test case), beta, and delta wolves guiding the search. Other test cases (omega wolves) adjust their positions based on these leaders, targeting untested code regions effectively
5. Hybrid GWO: This enhanced version of GWO incorporates a local search mechanism to refine the best test cases (e.g., alpha wolf). After updating positions, the algorithm makes minor adjustments to input values to enhance coverage, focusing on edge cases and challenging-to-reach code paths. This improves exploitation, making it more effective for complex logic.

The process is initiated by accessing and analyzing the cleaned dataset for feature extraction. A JavaScript script, integrated with the Vue.js framework, is employed to facilitate this task. The script is executed as a web application to search for directory names such as "epoch 1" and "epoch 2," thereby locating the corresponding result files. Upon identifying a matching directory, the script parses the HTML from result files (e.g., "result.txt") to extract data features, including input triples (a, b, c) and coverage metrics. In instances where no directory match is initially detected, the script conducts an iterative search for a matching file. After extraction, the data is processed by the Vue.js application to calculate the features described in Section 3.5, and a comprehensive table is generated that incorporates all features, including line coverage and branch coverage. Two distinct datasets are subsequently constructed for machine learning applications: one with line coverage designated as the dependent variable and the other with branch coverage in that role.

A Decision Tree is a non-parametric supervised learning model that predicts the value of a target variable by learning simple decision rules inferred from data features. It functions by building a tree-like structure where the data is recursively partitioned into smaller subsets. Each internal node represents a decision based on a specific feature, each branch represents the outcome of that decision, and each terminal node (or leaf) represents a final class label. We selected the DT classifier for this study for several key reasons. Its high interpretability is a primary advantage, as the transparent, rule-based structure makes it easy to understand how test data features influence classification. DTs can also naturally handle non-linear relationships and do not require extensive data preprocessing like feature scaling. Furthermore, their computational efficiency makes them suitable for our experimental setup, and they serve as an excellent baseline against which more complex models can be compared.

A Support Vector Machine is a powerful supervised learning algorithm used for classification. Its fundamental principle is to find an optimal hyperplane in a high-dimensional space that distinctly

separates data points of different classes. The "optimal" hyperplane is the one that maximizes the margin, the distance to the nearest data points of any class, which enhances the model's generalization capabilities.We chose SVM for this study because of its effectiveness in high-dimensional spaces, which is relevant given our large number of engineered features. It is also highly robust against overfitting, particularly in cases where the number of features is large relative to the number of samples. By using different kernel functions (e.g., linear, RBF), SVMs can model complex, non-linear decision boundaries, providing a powerful alternative to the rule-based approach of Decision Trees.

A Multilayer Perceptron is a class of feedforward artificial neural network (ANN). It consists of an input layer, one or more hidden layers of neurons, and an output layer. Each neuron in a layer is connected to all neurons in the subsequent layer, and it processes inputs using a nonlinear activation function. This architecture allows the MLP to learn and model complex, non-linear patterns from the data through a process called backpropagation, where the model's internal weights are adjusted to minimize prediction error. The MLP was included in our selection to represent a more complex, "black-box" modeling approach. Its primary strength is its ability to learn intricate relationships between features that other models might miss, potentially leading to higher predictive accuracy. By comparing its performance to more interpretable models like DT, we can evaluate the trade-off between model complexity and interpretability for our specific task.

The Naive Bayes classifier is a simple but effective probabilistic algorithm based on Bayes' theorem. It operates on the "naive" assumption that all input features are conditionally independent of one another, given the class label. Despite this simplification, the model calculates the probability of a data point belonging to each class and assigns it to the class with the highest probability. We selected Naive Bayes primarily for its computational efficiency and its value as a strong probabilistic baseline. It is extremely fast to train, making it ideal for rapid experimentation. Its performance provides a benchmark that helps us understand whether more complex models are offering a significant advantage over a simpler, probability-based approach on our dataset.

The focus of this section is the evaluation methodology for the machine learning (ML) classifiers. A rigorous assessment is paramount; therefore, our experimental design is founded upon a robust set of validation techniques. The evaluation criteria outlined in the methodology are systematically applied to assess the performance of machine learning (ML) classifiers, including Decision Tree, Support Vector Machine (SVM), Multilayer Perceptron (MLP), and Naive Bayes, on a dataset of 1004 labeled test suites. These criteria, designed to ensure a thorough analysis, include 10-fold cross-validation, confusion matrix collection, learning curve analysis, and a suite of performance metrics tailored to address the dataset's imbalanced nature and the research objectives outlined in Chapter 1.The 10-fold cross-validation process begins by dividing the 1004 test suites into 10 approximately equal folds (around 100–101 instances each), with stratified sampling preserving the class imbalance (e.g., 40.2% Class A in branch coverage, 34.1% Class E in line coverage). Each model is trained on nine folds and validated on the remaining fold, repeated 10 times, with each fold serving as the validation set once. To enhance reliability, this process is repeated five times with different random seeds (e.g., 0 through 4, as seen in the Decision Tree code), resulting in 50 validation iterations. The averaged performance metrics, such as accuracy, precision, recall, F1-score, and AUC, are derived from these runs. Confusion matrices are collected for each validation fold, forming a 5×5 matrix for the five classes (A to E) to track true versus predicted labels. The 50 matrices (10 folds × 5 runs) are averaged to identify misclassification patterns, such as confusion between Class B (61–80%)

and Class C (41–60%), which inform model tuning. Learning curve analysis evaluates model behavior as the training size increases, from 10% (100 instances) to 100% (1004 instances) of the dataset. Using the scikit-learn learning-curve function, training and validation accuracies are plotted for each model.

The trained and evaluated classifier is deployed in the final phase of the methodology as an operational tool for prioritizing test cases within the framework. Class labels (A, B, C, D, or E) are generated by the model, and these classifications are inherently mapped to priority levels to facilitate a structured ranking.

An ordered list of test cases is generated to optimize early fault detection and maximize code coverage. The prioritization procedure is structured such that a set of new or existing test cases is subjected to the feature extraction process described in Section 3.5 for the derivation of pertinent features. The classifier with superior performance, as determined through the evaluation in Section 3.8, is utilized to predict the quality class for each test case. Ranking of the test cases is then performed in descending order of predicted class quality, beginning with all Class A test cases, followed by Class B, and continuing sequentially to Class E. Within each class, sequencing of test cases may be conducted randomly or refined through a secondary metric, such as execution time.

This prioritized list of test cases serves as the culmination of the framework, which can be executed in sequence, starting from the highest-priority items. Overall test execution time and costs are thereby curtailed through the strategic emphasis on the most efficacious test cases at the outset, advancing the research objectives directly.

Results

Accuracy of Predictive Models

Table 1 and 2 present the best, worst, and average accuracy values across these runs for line coverage and branch coverage datasets, respectively.

Table 1. Accuracy of ML Models on the Line Coverage Dataset

Model	Best Accuracy	Worst Accuracy	Average Accuracy
Decision Tree	0.9731	0.9731	0.9731
Naive Bayes	0.8865	0.8855	0.886
Multi-Layer Perceptron	0.9452	0.9233	0.9372
Support Vector Machine	0.9522	0.9482	0.9506

Table 2. Accuracy of ML Models on Branch Coverage Dataset

Model	Best Accuracy	Worst Accuracy	Average Accuracy
Decision Tree	0.8844	0.8745	0.8791
Naive Bayes	0.8258	0.8217	0.8245
Multi-Layer Perceptron	0.8736	0.8575	0.8651
Support Vector Machine	0.8745	0.8595	0.8659

The Decision Tree stands out with near-perfect accuracy on the line coverage dataset, while all models show a drop on the branch coverage dataset, reflecting its greater complexity.

Precision of Predictive Models

Tables 3 and 4 summarize the best, worst, and average precision values for ML models, evaluated on the line and branch coverage datasets, respectively.

Table 3. Precision of ML Models on Line Coverage Dataset

Model	Best Precision	Worst Precision	Average Precision
Decision Tree	0.9765	0.9716	0.9752
Naive Bayes	0.8413	0.8401	0.8408
Multi-Layer Perceptron	0.8728	0.705	0.813
Support Vector Machine	0.925	0.8945	0.9133

Table 4. Precision of ML Models on Branch Coverage Dataset

Model	Best Precision	Worst Precision	Average Precision
Decision Tree	0.7991	0.7882	0.7933
Naive Bayes	0.6289	0.6265	0.6277
Multi-Layer Perceptron	0.7866	0.7453	0.761
Support Vector Machine	0.7874	0.75	0.7721

Sensitivity (Recall) of Predictive Models

Tables 5 and 6 present the best, worst, and average recall scores for the evaluated models across the line and branch coverage datasets, respectively.

Table 5. Recall of ML Models on Line Coverage Dataset

Model	Best Recall	Worst Recall	Average Recall
Decision Tree	0.9046	0.9022	0.9037
Naive Bayes	0.9294	0.9288	0.9291
Multi-Layer Perceptron	0.8292	0.7356	0.788
Support Vector Machine	0.8319	0.8211	0.8281

Table 6. Recall of ML Models on the Branch Coverage Dataset

Model	Best Recall	Worst Recall	Average Recall
Decision Tree	0.7801	0.7588	0.769
Naive Bayes	0.7433	0.7342	0.74
Multi-Layer Perceptron	0.7649	0.7351	0.7477
Support Vector Machine	0.772	0.7424	0.7563

F1-Score of Predictive Models

Tables 7 and 8 show the best, worst, and average F1-scores for each classifier trained on the line and branch coverage datasets.

Table 7. F1-Score of ML Models on Line Coverage Dataset

Model	Best F1-Score	Worst F1-Score	Average F1-Score
Decision Tree	0.9291	0.9232	0.9267
Naive Bayes	0.8422	0.841	0.8417
Multi-Layer Perceptron	0.8347	0.7185	0.7862
Support Vector Machine	0.8521	0.8326	0.8455

Table 8. F1-Score of ML Models on Branch Coverage Dataset

Model	Best F1-Score	Worst F1-Score	Average F1-Score
Decision Tree	0.7801	0.7564	0.7676
Naive Bayes	0.6591	0.6555	0.6577
Multi-Layer Perceptron	0.7578	0.7282	0.7383
Support Vector Machine	0.7665	0.7361	0.7521

Training Curves

To evaluate the training behaviour and generalization performance of each predictive model, learning curves were generated for both the line and branch coverage datasets. Each learning curve plots

the training and validation accuracy over the number of training iterations or sample sizes. These curves provide insight into whether a model is suffering from overfitting, underfitting, or is well-generalized.

Learning Curves for Line Coverage Dataset

This subsection presents the learning curve results of each ML model trained and evaluated using the Line Coverage dataset. Each figure is followed by a brief interpretation of the model's generalization behavior.

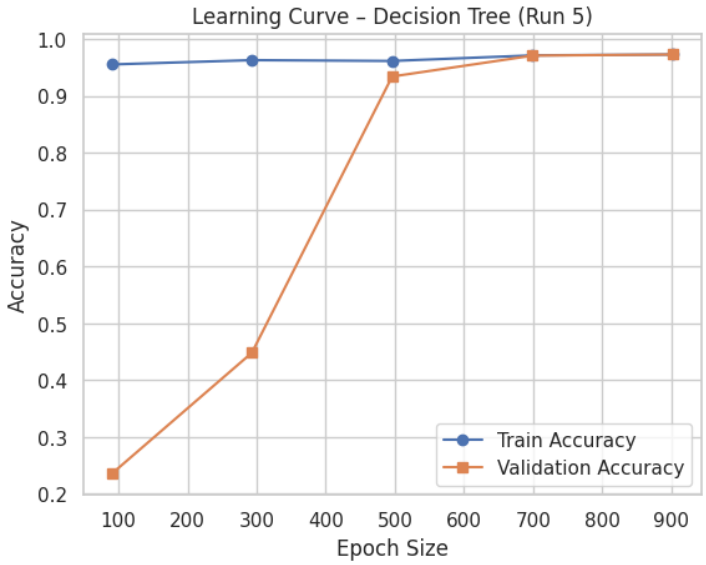

Figure 1. Learning Curve - DT (Line Coverage)

As shown in Figure 1, the learning curve clearly shows the Decision Tree's tendency to overfit on smaller datasets, as evidenced by the large initial gap between training and validation accuracy. However, this overfitting is overcome with more data, as the validation accuracy rapidly improves and converges with the training accuracy, indicating the model generalizes well when provided with a sufficiently large training set.

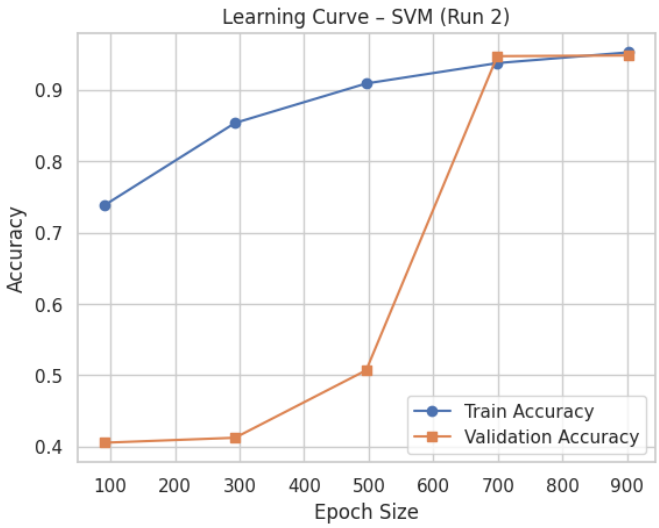

Figure 2. Learning Curve - SVM (Line Coverage)

As shown in Figure 2, the learning curve for the SVM model reveals a clear progression from underfitting to a well-generalized state. Initially, with smaller training sets, both training and validation accuracies are suboptimal, indicating the model is underfitting and has not yet captured the underlying

data patterns. However, as the training set size increases, the validation accuracy shows a steady and significant improvement, eventually converging with the training accuracy. This narrowing gap demonstrates that the model benefits greatly from more data, overcoming its initial limitations to generalize effectively. This makes SVM a strong candidate for this task, particularly with large test datasets.

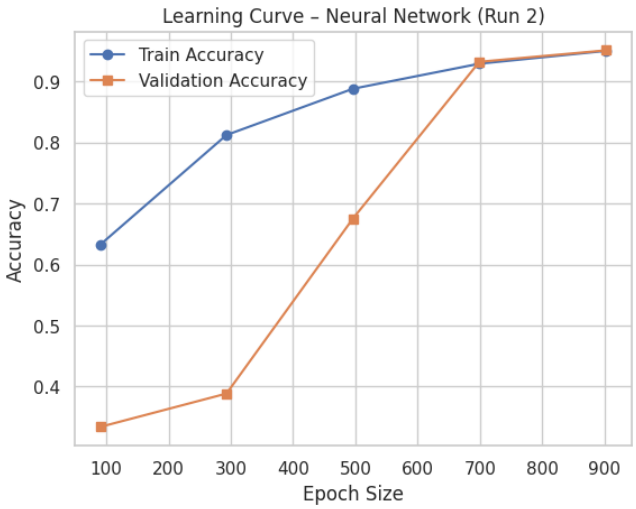

Figure 3. Learning Curve – MLP (Line Coverage)

As shown in Figure 3, the MLP model exhibits a powerful learning capability, with both training and validation accuracy increasing consistently as the training set size grows. The two curves track each other closely, confirming the model's ability to learn complex, non-linear patterns from the data. A small, persistent gap between the curves at the final stage indicates a minor degree of overfitting, which is common for highly complex models, such as MLP. Nevertheless, its high performance on both sets validates its use for test prioritization tasks where capturing subtle feature interactions is critical for accurate classification.

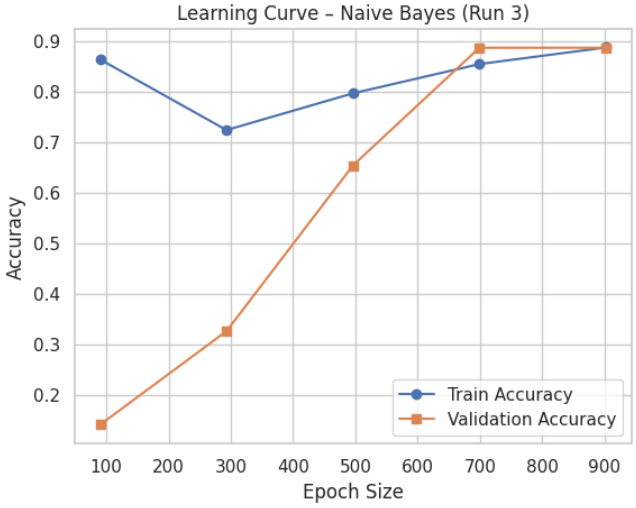

Figure 4. Learning Curve – NB (Line Coverage)

As shown in Figure 4, the Naive Bayes model initially struggles with smaller datasets, exhibiting classic signs of underfitting where both training and validation accuracies are moderate. However, the model shows significant improvement with more data. The validation accuracy rises substantially as the training set grows, eventually converging with the training accuracy score. This suggests that, although the model may be too simplistic for sparse data, it becomes a reliable and well-generalized classifier once

a sufficient volume of data is available. Its computational efficiency makes it a viable lightweight solution for this classification task.

Learning Curves for Branch Coverage Dataset

This subsection includes the learning curves for all models when applied to the Branch Coverage dataset. The curves demonstrate how training and validation accuracy evolve with increasing training sizes, highlighting patterns of overfitting or underfitting.

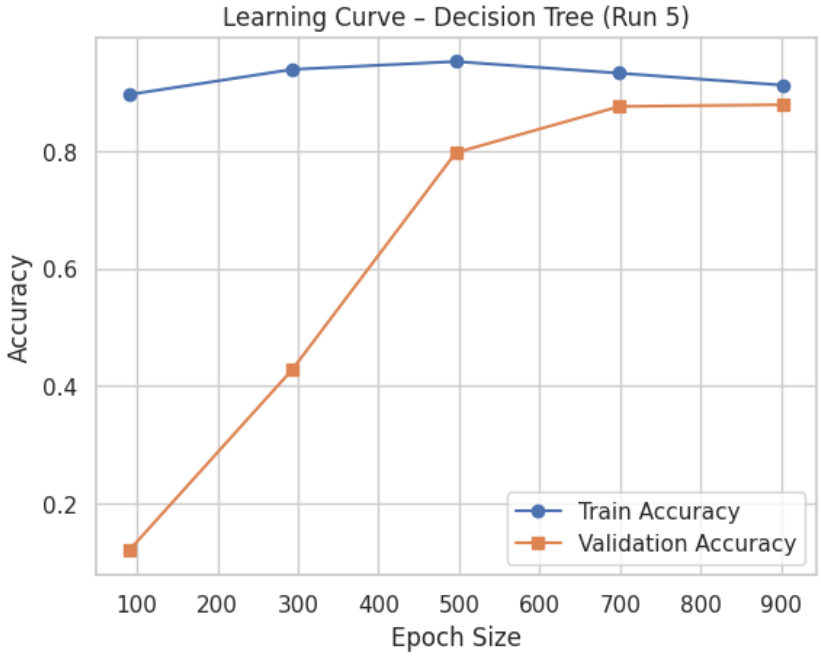

Figure 5. Learning Curve – DT (Branch Coverage)

As shown in Figure 5, the Decision Tree model demonstrates a highly effective learning process on the branch coverage dataset. The validation accuracy experiences rapid growth, closely tracking the high training accuracy from the early stages. The two curves quickly converge and stabilize, with a minimal gap between them. This behavior indicates a well-fitted model that generalizes effectively without significant overfitting or underfitting. The model's ability to achieve high performance so efficiently makes it a robust and reliable candidate for prioritization tasks involving structured feature sets, such as branch coverage.

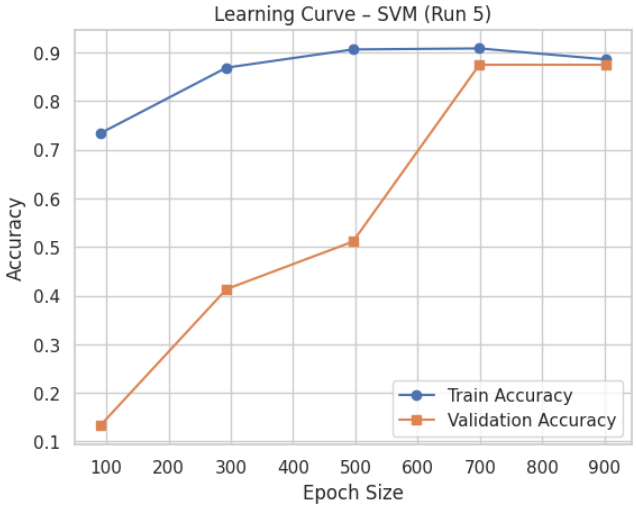

Figure 6. Learning Curve – SVM on Branch Coverage

The learning curve for the SVM model, presented in Figure 6, illustrates a steady and consistent learning pattern. Both the training and validation accuracy curves rise in parallel, indicating that the model generalizes well at each stage as more data is introduced. The consistent, narrow gap between the curves suggests the model has a low variance and is not prone to overfitting on this dataset. This reliable performance confirms the SVM's suitability for classifying complex patterns within test data, making it a dependable choice for prioritizing branch coverage.

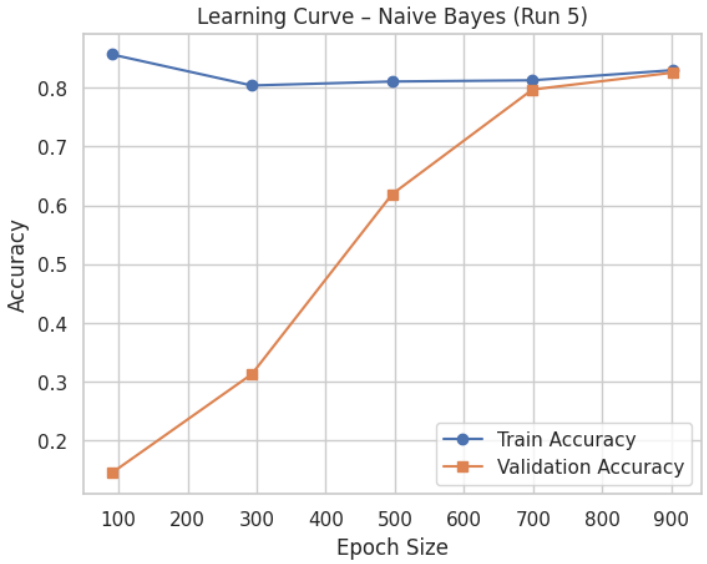

Figure 7. Learning Curve – NB (Branch Coverage)

As illustrated in Figure 7, the Naive Bayes model initially exhibits signs of high variance, characterized by a wide gap between its high training accuracy and low validation accuracy. This suggests the model overfits on smaller training sets. However, as the training data increases, the validation accuracy improves sharply and converges towards the training score. This demonstrates that the model's initial overfitting is overcome with sufficient data, leading to a balanced and well-generalized final performance. The model's computational efficiency, combined with its eventual stability, makes it a practical choice for fast and scalable testing scenarios.

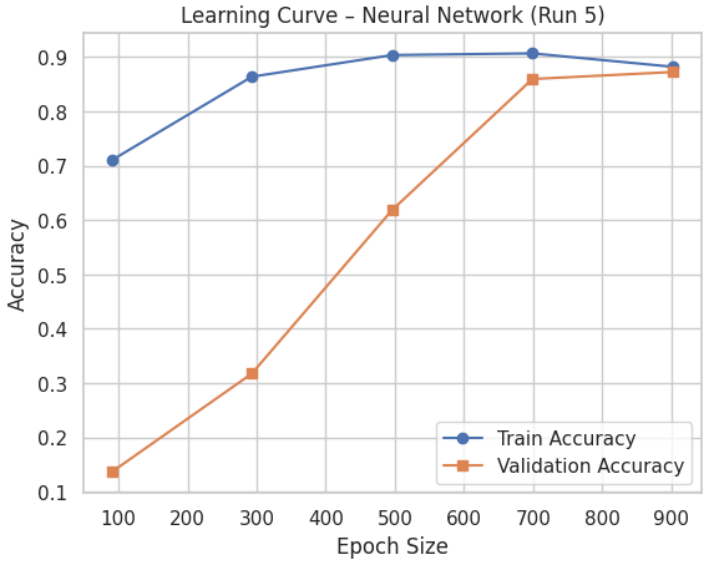

Figure 8. Learning Curve – MLP on Branch Coverage

The learning curve for the MLP model, shown in Figure 8, highlights its strong capacity for learning from the branch coverage data. Both the training and validation curves show a pronounced and synchronized rise in accuracy. The close convergence of the two curves towards the end of the training process indicates excellent generalization with minimal overfitting. This performance highlights the MLP's ability to handle the complex, potentially non-linear relationships inherent in branch coverage data, making it a highly suitable model for achieving accurate test case prioritization.

Discussion and Conclusion

This section provides a detailed analysis and interpretation of the experimental results presented earlier, organized around key themes: overall model performance, the impact of coverage criteria, comparison with existing work, and responses to the research questions. Additional implications and limitations are explored to highlight the framework's potential.

The results, reveal clear performance tiers among the evaluated models. The decision tree consistently excels, achieving the highest accuracy (97.31% on line coverage), precision (97.52%), and F1-score (92.67%), likely due to its ability to leverage rule-based relationships within the 64+ features, such as Coverage TDs Average and Cyclomatic Complexity. This strength makes it particularly effective for structured datasets, such as those generated from the triangle classification benchmark. SVM and MLP also perform well, with SVM achieving 95.06% accuracy and MLP capturing complex, non-linear patterns, which suggests their utility in scenarios with intricate feature interactions. However, Naive Bayes lags, especially on the branch coverage dataset (82.45% accuracy, 62.77% precision), reflecting its limitation in handling correlated features, a common trait in software metrics. The Decision Tree's dominance suggests it could be a go-to choice for similar prioritization tasks in software testing, especially where interpretability is valued alongside high performance. The strong showing of SVM and MLP indicates that the framework can adapt to more complex projects by selecting models based on data characteristics. This flexibility enhances the framework's applicability across different software development contexts. The Decision Tree's reliance on rule-based learning may struggle with highly dynamic or unstructured codebases where non-linear patterns dominate. Naive Bayes' underperformance highlights a potential weakness when feature dependencies are strong, limiting its use unless preprocessing can reduce correlations. Additionally, the computational cost of training an MLP, though not quantified here, may pose challenges for large-scale deployments, a factor to consider in resource-constrained environments.

A significant finding is the performance disparity between the line and branch coverage datasets. All models perform better on line coverage (e.g., Decision Tree at 97.31% accuracy) than branch coverage (87.91% accuracy), indicating that line coverage prediction is a more straightforward task. Naive Bayes experiences the most significant drop (from 84.08% to 62.77% precision), underscoring its difficulty with the complex logical paths in branch coverage. The Decision Tree and SVM maintain relative stability, demonstrating their adaptability to these challenges. The superior performance on line coverage suggests the framework could be particularly effective in projects where line-level testing is prioritized, such as unit testing in early development phases. The stability of Decision Trees and SVMs across datasets suggests that they can accommodate diverse coverage requirements, providing a versatile solution for teams balancing various testing goals. The lower performance on branch coverage highlights a potential limitation in detecting faults tied to control flow, which is critical for integration testing. The framework

may require additional features or model adjustments (e.g., deeper trees or advanced kernels) to address this gap. Furthermore, the reliance on the triangle benchmark's specific structure might not fully generalize to programs with more nested or dynamic branching, warranting further testing.

To contextualize the contributions of this research, we compared the performance of our proposed framework with the results reported in a recent study by Sharma et al. (2024), Kiran et al. (2025), and Son et al. (2025). Our Decision Tree model achieves an outstanding accuracy of 0.9731 and precision of 0.9752 on the Triangle benchmark dataset, significantly outperforming Sharma et al.(2024) model (0.6165 accuracy, 0.21 precision), Kiran et al.(2025) Random Forest (0.79 accuracy, 0.80 precision), and Son et al. (2025) CatBoost (0.96 accuracy, 0.27 precision). The high precision of our model reflects its ability to prioritize high-impact test cases with minimal false positives.

The Decision Tree's recall (0.9037) and F1-score (0.9267) surpass those of Sharma et al. (0.61 recall, 0.58 F1-score), Kiran et al. (0.82 recall, 0.81 F1-score), and Son et al. (0.29 recall, 0.28 F1-score). This demonstrates a robust balance of fault detection and prioritization efficiency, making our framework highly effective for balanced TCP. Unlike Son et al. (2025) study, which is skewed by a 97% pass class, our evaluation employs stratified 10-fold cross-validation on a balanced dataset, ensuring reliable and generalizable results. The significant improvement over Sharma et al.'s low precision (0.21) and F1-score (0.58) highlights the strength of our metaheuristic-driven test data generation and comprehensive feature set (e.g., Coverage TDs Average, Cyclomatic Complexity).

Several recommendations arise from the study's outcomes to further refine the framework and extend its scope. Deep learning algorithms could be explored to manage intricate patterns in test data, potentially elevating accuracy on complex branch coverage scenarios. Reinforcement learning methods might also be applied to enable dynamic prioritization as new faults emerge, building upon the current metaheuristic foundation.A user-friendly tool could be developed to incorporate the trained models, enabling testers to input test case details and generate prioritized lists. Expanding the dataset with additional test cases from varied benchmarks would bolster generalizability, while integrating with continuous integration pipelines could automate prioritization in real-time workflows.These suggestions target identified limitations, such as dependence on generated data quality and performance on the line and branch code, promoting broader practical adoption.

References

Arasteh, B., Hosseini, S.M.J. (2022). Traxtor: An Automatic Software Test Suit Generation Method Inspired by Imperialist Competitive Optimization Algorithms. *Journal of Electronic Testing*, 205-215.

Bertolino, A., Guerriero, A., Miranda, B., Pietrantuono, R., & Russo, S. (2020). Learning-to-rank vs ranking-to-learn: Strategies for regression testing in continuous integration. 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), Seoul, Korea (South).

Khambam, S. K. R., Kaluvakuri, V. P. K., & Peta, V. P. (2022). Optimizing Cloud-Based Regression Testing: A Machine Learning-Driven Paradigm for Swift and Effective Releases. *Available at SSRN 4927238*.

Konidena, B. K., Bairi, A. R., & Pichaimani, T. (2021). Reinforcement Learning-Driven Adaptive Test Case Generation in Agile Development. *American Journal of Data Science and Artificial Intelligence Innovations*, *1*, 241-273.

Lachmann, R. (2018). Machine learning-driven test case prioritization approaches for black-box software testing. The European test and telemetry conference, Nuremberg, Germany,

Lonetti, F., & Marchetti, E. (2018). Emerging Software Testing Technologies. In. Elsevier.

Marijan, D. (2023). Comparative study of machine learning test case prioritization for continuous integration testing. *Software Quality Journal*.

Muhammad, A. (2024). AI-Driven Testing Automation: Harnessing Machine Learning for Intelligent Test Case Creation and Predictive Defect Analysis. *International Journal of Artificial Intelligence and Applications*, *9*(3), 22-35.

Mukherjee, R., & Patnaik, K. S. (2021). A survey on different approaches for software test case prioritization *Journal of King Saud University – Computer and Information Sciences* 1041–1054 1040pl.

Pandhare, H. V. (2025). Future of software test automation using ai/ml. *International Journal Of Engineering And Computer Science*, *13*(05).

Ramachandran, S. (2026). Leveraging AI-driven multi-agents for next-generation software testing: a lattice-based cross-industry automation framework. *International Journal of Information Technology*, 1-8.

Ramzan, H. A., Islam, K., Hussain, M. A., Monim, R. M., Asad, S. M., & Ramzan, S. (2026). Data-Driven Test Case Prioritization (DD-TCP): A Machine Learning Framework for Intelligent Software Quality Assurance. *Computers, Materials, & Continua*, *88*(1).

Sakhrawi, Z., Labidi, T. (2024). Test case selection and prioritization approach for automated regression testing using ontology and COSMIC measurement. *Automated Software Engineering*.

Sivaraman, H. (2020). *Machine learning for software quality and reliability: Transforming software engineering*. Libertatem Media Private Limited.

Sugave, S. R., Kulkarni, Y.R., Jagdale, B. et al. (2025). Fault-Aware Test Case Prioritization in Software Testing Using Jaya Archimedes Optimization Algorithm. *Journal of Electron Test*.

Wang, X., Zhang, S. (2023). Cluster-based adaptive test case prioritization. *Information and Software Technology*, 107339.