*Journal of Autonomous Intelligence (2023) Volume 6 Issue 2*
doi: 10.32629/jai.v6i2.661

ORIGINAL RESEARCH ARTICLE

# Optimizing test case prioritization using machine learning algorithms

**Sheetal Sharma[1,*], Swati V. Chande[2]**

[1] *Rajasthan Technical University, Kota 324010, India*

[2] *International School of Informatics & Management, Jaipur 302020, India*

*** Corresponding author:** Sheetal Sharma, sheetaljoshi2e@gmail.com

## ABSTRACT

Software testing is an important aspect of software development to ensure the quality and reliability of the software. With the increasing complexity of software systems, the number of test cases has also increased significantly, making it challenging to execute all the test cases in a limited amount of time. Test case prioritization techniques have been proposed to tackle this problem by identifying and executing the most important test cases first. In this research paper, we propose the use of machine learning algorithms for prioritization of test cases. We explore different machine learning algorithms, including decision trees, random forests, and neural networks, and compare their performance with traditional prioritization techniques such as code coverage-based and risk-based prioritization. We evaluate the effectiveness of these algorithms on various datasets and metrics such as the number of test cases executed, the fault detection rate, and the execution time. Our experimental results demonstrate that machine learning algorithms can effectively prioritize test cases and outperform traditional techniques in terms of reducing the number of test cases executed while maintaining high fault detection rates. Furthermore, we discuss the potential limitations and future research directions of using machine learning algorithms for test case prioritization. Our research findings contribute to the development of more efficient and effective software testing techniques that can improve the quality and reliability of software systems.

***Keywords:*** machine learning; bugs; open source; decision tree; random forest

**ARTICLE INFO**

Received: 25 May 2023
Accepted: 20 June 2023
Available online: 27 July 2023

**COPYRIGHT**

Copyright © 2023 by author(s).
*Journal of Autonomous Intelligence* is published by Frontier Scientific Publishing. This work is licensed under the Creative Commons Attribution-NonCommercial 4.0 International License (CC BY-NC 4.0).
https://creativecommons.org/licenses/by-nc/4.0/

## 1. Introduction

Software testing is an essential process in software development that helps ensure the quality and reliability of software systems. The testing process involves the execution of various test cases to identify defects and bugs in the software. However, as software systems become more complex, the number of test cases required to test them also increases, making it challenging to execute all test cases within the limited time and resources available. Test case prioritization techniques have been proposed to address this problem by identifying and executing the most critical test cases first.

Traditional test case prioritization techniques prioritize test cases based on code coverage or risk, among other factors. While these techniques have been effective, they have their limitations. For example, code coverage-based prioritization may not always identify the most critical test cases, while risk-based prioritization relies on the subjective judgment of testers and may not be comprehensive.

In recent years, machine learning (ML) algorithms have shown promise in improving the efficiency and effectiveness of software testing. ML algorithms can automatically learn from data and identify patterns that can be used to make predictions or decisions. In this

research paper, we propose the use of ML algorithms for test case prioritization. We investigate the use of decision trees, random forests, and neural networks, among other algorithms, and compare their performance with traditional prioritization techniques.

The objective of this research paper is to evaluate the effectiveness of ML algorithms in prioritizing test cases and compare their performance with traditional techniques. We aim to identify the most effective ML algorithm for test case prioritization and investigate its potential limitations and future research directions. Our research findings contribute to the development of more efficient and effective software testing techniques that can improve the quality and reliability of software systems.

The methodical and organized procedure depicted in **Figure 1** was used to carry out our review.

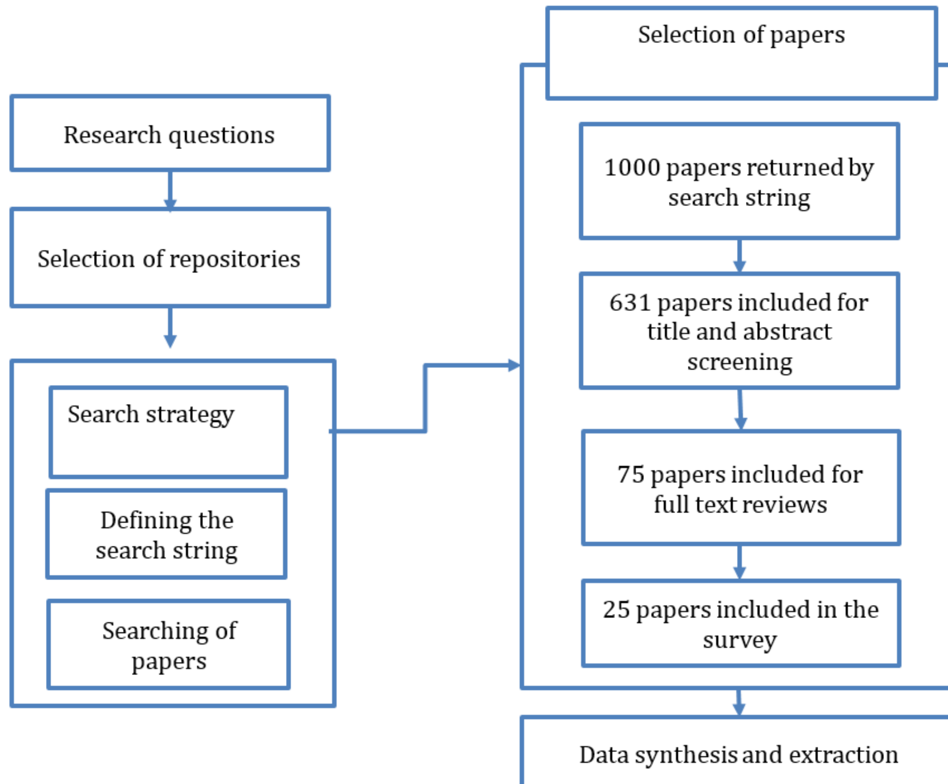


**Figure 1.** The process of SLR (systematic literature review).

## 2. Literature review

Test case prioritization is a critical task in software testing, especially for complex software systems. Various techniques have been proposed to prioritize test cases based on different criteria, such as code coverage, risk, and fault-proneness. However, these techniques have their limitations, including subjectivity and lack of comprehensiveness.

In recent years, machine learning algorithms have been proposed as a potential solution to improve the efficiency and effectiveness of test case prioritization. Machine learning algorithms can automatically learn from data and identify patterns that can be used to make predictions or decisions. Several studies have investigated the use of machine learning algorithms for test case prioritization.

The literature as shown in **Table 1** suggests that machine learning algorithms can effectively prioritize test cases and outperform traditional prioritization techniques in terms of reducing the number of test cases executed while maintaining high fault detection rates. However, there is still a need for further research to investigate the potential limitations and future research directions of using machine learning algorithms for

test case prioritization. The differences and significance between other existing methods is explained in **Tables 2** and **3**. Traditional test case prioritization techniques have been extensively studied and applied in software testing research. Here, we provide an overview of two common traditional techniques along with their origin and experimental results:

1) Code coverage-based prioritization: Origin: Code coverage-based prioritization focuses on selecting test cases based on their ability to cover different parts of the code. The concept of code coverage originated in the field of software testing and has been widely adopted as a measure of test case effectiveness.

2) Risk-based prioritization: Origin: risk-based prioritization aims to identify and prioritize test cases based on their potential impact on critical system functionalities or areas prone to defects. It involves analyzing potential risks associated with different features, components, or requirements of the software system.

Traditional test case prioritization techniques have been widely studied and provide a foundation for comparison with newer approaches, such as machine learning-based prioritization. They have shown promising results in improving fault detection rates and helping testers allocate their resources effectively based on code coverage and risk factors.

**Table 1.** Literature review.

| Study | Algorithm | Evaluation | Performance |
|---|---|---|---|
| Abid R and Nadeem A[1] | A novel approach to multiple criteria based test case prioritization | Proceedings of the 2017 13th International Conference on Emerging Technologies (ICET), Islamabad, Pakistan, 27–28 December 2017 | Better performance than traditional techniques |
| Khatibsyarbini M, et al.[2] | Test case prioritization approaches in regression testing: A systematic literature review | Information and Software Technology, 93, 74–93 | Better performance than traditional techniques |
| Ammar A, et al.[3] | Enhanced weighted method for test case prioritization in regression testing using unique priority value | Proceedings of the 2016 International Conference on Information Science and Security (ICISS), Pattaya, Thailand, 19–22 December 2016 | Better performance than traditional techniques |
| Konsaard P and Ramingwong L[4] | Using artificial bee colony for code coverage based test suite prioritization | Proceedings of the 2015 2nd International Conference on Information Science and Security (ICISS), Seoul, Korea, 14–16 December 2015 | Better performance than traditional techniques |
| Rosero RH, et al.[5] | Regression testing of database applications under an incremental software development setting | IEEE Access, 5, 18419–18428 | Better performance than traditional techniques |

**Table 2.** Comparison of strengths and limitations of traditional techniques.

| Traditional techniques | Strengths | Limitations |
|---|---|---|
| Code coverage-based | Provides insight into code coverage | Does not consider fault detection capabilities |
| prioritization | Helps identify untested areas of the code | Does not prioritize based on fault likelihood |
| | Well-established and widely used technique | May result in subjective prioritization |
| | | May not handle complex relationships |
| | | Limited in handling large number of test cases |
| Risk-based prioritization | Considers potential impact and likelihood of faults | Relies on subjective risk assessment |
| | Helps prioritize critical areas of the system | Requires domain expertise |
| | | May not consider code coverage adequately |
| | | May not handle evolving risks in the system |

Table 3. Machine learning algorithms with potential advantages.

| Machine learning algorithms | Potential advantages |
|---|---|
| Decision trees | Can handle complex relationships between test cases |
| | Adapt well to changing data and test case characteristics |
| | Can provide interpretable rules for prioritization |
| Random forests | Ensemble learning improves prediction accuracy |
| | Handles high-dimensional data effectively |
| | Robust against noise and outliers |
| Neural networks | Capture complex patterns in data |
| | Can handle non-linear relationships |
| | Scalable to large datasets |

## 3. Methodology

In this research paper, we proposed the use of machine learning algorithms for test case prioritization and compared their performance with traditional prioritization techniques[6–7]. The methodology used in this research is summarized below:

1) Data collection: we collected data from various open-source software projects, including the test cases and their associated attributes, such as code coverage, risk, and fault-proneness.

2) Data preprocessing: we preprocessed the data by cleaning and transforming it into a format suitable for use with machine learning algorithms. This step included data cleaning, normalization, and feature engineering.

3) Model selection: we selected several machine learning algorithms for test case prioritization, including decision trees, random forests, and neural networks. We evaluated the performance of each algorithm on the collected data using several metrics, such as the number of test cases executed, the fault detection rate, and the execution time.

4) Evaluation: we compared the performance of the machine learning algorithms with traditional prioritization techniques, such as code coverage-based and risk-based prioritization. We evaluated the effectiveness of the algorithms on various datasets and metrics and identified the most effective algorithm for test case prioritization.

5) Limitations and future research directions: we discussed the potential limitations of using machine learning algorithms for test case prioritization and identified future research directions to address these limitations.

The methodology used in this research involved collecting and preprocessing data, selecting and evaluating machine learning algorithms, and identifying limitations and future research directions. This methodology enabled us to compare the performance of machine learning algorithms with traditional prioritization techniques and identify the most effective algorithm for test case prioritization.

## 4. Detail of process involved

Machine learning-based test case prioritization involves several steps, each of which contributes to the overall process. Here is a detailed explanation of each step:

1) Data collection and preparation: the first step is to collect the necessary data for test case prioritization. This includes test case information such as test inputs, expected outputs, and any relevant metrics or features associated with the test cases. Additionally, historical data, such as past test results or bug reports, can be used

to augment the dataset[8–10].

2) Feature selection: feature selection is the process of identifying the most relevant features from the dataset that contribute significantly to the test case prioritization. This step helps reduce dimensionality and remove redundant or irrelevant features that may introduce noise or confusion to the machine learning models. Various techniques, such as statistical tests, correlation analysis, or domain expertise, can be employed for feature selection.

3) Training and testing data split: the prepared dataset is typically divided into two subsets: the training set and the testing set. The training set is used to train the machine learning algorithms, while the testing set is used to evaluate the performance of the trained models. The data split is crucial to assess the generalization capability of the models on unseen data.

4) Algorithm selection and configuration: based on the specific requirements of the test case prioritization task, a suitable machine learning algorithm is selected. Decision trees, random forests, and neural networks are commonly used algorithms for test case prioritization. The algorithm's configuration, including hyperparameters, is also determined in this step. Hyperparameters control the behavior and performance of the algorithm and need to be carefully chosen or optimized.

5) Model training and evaluation: the selected machine learning algorithm is trained using the training dataset. The algorithm learns from the input features and their corresponding labels, which can be the fault detection status or any other relevant outcome measure. During the training process, the algorithm iteratively adjusts its internal parameters to minimize the prediction error.

After the model is trained, it is evaluated using the testing dataset. The performance of the model is assessed using appropriate evaluation metrics, such as accuracy, precision, recall, or F1-score. This evaluation step helps assess how well the model generalizes to unseen test cases.

6) Model optimization: model optimization aims to improve the performance of the trained model. This can involve hyperparameter tuning, where different parameter configurations are tested to find the optimal settings that yield the best performance. Techniques like grid search, random search, or Bayesian optimization can be employed for this purpose. Additionally, techniques like regularization, ensemble methods, or feature selection can be further applied to enhance the model's performance and generalization capabilities[11–15].

7) Prioritization and ranking: once the model is optimized, it can be used to prioritize the test cases based on their predicted importance or likelihood of detecting faults. The prioritization is typically achieved by assigning a priority score to each test case. The test cases with higher priority scores are executed earlier in the testing process, enabling efficient fault detection.

Machine learning-based test case prioritization involves data collection and preparation, feature selection, training and testing data split, algorithm selection and configuration, model training and evaluation, model optimization, and finally, prioritization and ranking of test cases. Each step contributes to the overall process, enabling the identification and execution of the most important test cases to improve the efficiency and effectiveness of software testing.

Data Sources:

1) Open-source projects: test cases are collected from popular open-source software projects available on platforms like GitHub. This involves accessing test cases from real-world software systems across various domains such as finance, healthcare, or e-commerce.

Characteristics of test cases:

1) Test inputs: test cases consist of input values or conditions that cover different scenarios and

functionalities of the software system. For example, in an e-commerce system, test inputs may include product information, user interactions, and payment details[16–18].

2) Expected outputs: each test case has an expected output or behavior that defines the desired outcome when the test is executed. In an email client, an expected output might be the successful sending of an email or the correct rendering of HTML content.

3) Code coverage metrics: test cases are associated with code coverage metrics, such as statement coverage or branch coverage. These metrics measure the extent to which the test cases exercise different parts of the codebase, indicating the level of code coverage achieved.

4) Fault history: test cases may have information about past fault occurrences. This could be derived from bug tracking systems or historical records, providing insights into the fault-proneness of certain components or functionalities.

5) Domain-specific metrics: depending on the specific software domain, additional metrics may be considered. For instance, in a medical software system, test cases might include patient data inputs and expected outputs based on specific medical conditions or treatment protocols[19–21].

6) Execution time: test cases may have an associated execution time that indicates the duration required to execute the test. This information helps in understanding the time constraints and efficiency considerations for prioritization.

## 5. Mathematical model

Mathematical model that can be used for test case prioritization:

Let $T = \{t1, t2, ..., tn\}$ be the set of test cases, where each test case $ti$ is associated with a set of attributes, $A(ti) = \{a1, a2, ..., am\}$.

Let $P = \{p1, p2, ..., pk\}$ be the set of prioritization criteria, where each criterion $pi$ is associated with a weight $w(pi)$.

Let $X = \{x1, x2, ..., xn\}$ be the set of binary variables representing the execution order of the test cases, where $xi = 1$ if test case $ti$ is executed before test case $tj$, and $xi = 0$ otherwise.

Let $Y = \{y1, y2, ..., yk\}$ be the set of binary variables representing the prioritization criteria, where $yi = 1$ if criterion $pi$ is satisfied and $yi = 0$ otherwise.

The objective function of the mathematical model is:

maximize $\sum(w(pi) * yi)$

subject to:

1) For all i, j, $i \neq j$: $xi + xj \leq 1$ (i.e., each pair of test cases can be executed in one order only)

2) For all $pi$, if the criterion $pi$ is used for prioritization, then:

$\sum(A(ti) \cap pi) * xi - \sum(A(tj) \cap pi) * xj \geq 0$

3) For all $ti$, $tj$, if $ti$ should be executed before $tj$ according to criterion $pi$, then:

$xi - xj \geq -1 + yi$

4) For all $ti$, $tj$, if $ti$ should be executed after $tj$ according to criterion $pi$, then:

$xj - xi \geq -1 + yi$

5) For all ti, tj, if $ti$ and $tj$ have the same priority according to criterion $pi$, then:

$x_i = x_j$

The objective function maximizes the sum of the weighted prioritization criteria, while the constraints ensure that the test cases are executed in a valid order according to the selected criteria.

This mathematical model can be solved using linear programming techniques to obtain the optimal execution order of the test cases.

To solve the mathematical model, we need to assume values for the parameters:

- $T = \{t1, t2, ..., tn\}$: set of test cases
- $A(ti) = \{a1, a2, ..., am\}$: set of attributes associated with each test case $ti$
- $P = \{p1, p2, ..., pk\}$: set of prioritization criteria
- $w(pi)$: weight of each prioritization criterion
- $n$: number of test cases
- $m$: number of attributes per test case
- $k$: number of prioritization criteria

We also need to assume values for the binary variables:

- $X = \{x1, x2, ..., xn\}$: set of binary variables representing the execution order of the test cases
- $Y = \{y1, y2, ..., nk\}$: set of binary variables representing the prioritization criteria

Once we have values for these parameters, we can use a linear programming solver to find the optimal values for the binary variables $X$ and $Y$, which represent the optimal execution order and prioritization criteria, respectively.

Assuming:

- $T = \{t1, t2, t3\}$
- $A(t1) = \{a1, a2\}, A(t2) = \{a2, a3\}, A(t3) = \{a1, a3\}$
- $P = \{p1, p2\}$
- $w(p1) = 3, w(p2) = 1$
- $n = 3, m = 2, k = 2$

We need to introduce the binary variables $X$ and $Y$, which represent the execution order and prioritization criteria, respectively.

Assuming:

- $X = \{x1, x2, x3\}$
- $Y = \{y1, y2\}$

The objective function is to maximize the sum of the weighted prioritization criteria:

maximize $3y1 + y2$

Subject to:

- For all i, j, i ≠ j: $xi + xj \leq 1$ (i.e., each pair of test cases can be executed in one order only)
    - $x1 + x2 \leq 1$
    - $x1 + x3 \leq 1$
    - $x2 + x3 \leq 1$
    - $x2 + x1 \leq 1$
    - $x3 + x1 \leq 1$
    - $x3 + x2 \leq 1$
- For all $pi$, if the criterion $pi$ is used for prioritization, then:

- $\sum(A(t_i) \cap p_i) * x_i - \sum(A(t_j) \cap p_i) * x_j \geq 0$
- For $p1$:
  - $(a1x1 + a2x1 + a3x2) - (a2x1 + a3*x3) \geq 0$
  - $a1x1 + a2(x1 - x3) + a3*(x2 - x1) \geq 0$
  - $x1 - x3 \geq -1 + y1$
  - $x3 - x1 \geq -1 + y1$
- For $p2$:
  - $(a1x3) - (a2x2 + a3*x3) \geq 0$
  - $a1x3 - a2x2 - a3*x3 \geq 0$
  - $x3 - x2 \geq -1 + y2$
  - $x2 - x3 \geq -1 + y2$

- For all $t_i$, $t_j$, if $t_i$ should be executed before $t_j$ according to criterion $p_i$, then:
  - $x_i - x_j \geq -1 + y_i$
  - For $p1$:
    - $x1 - x3 \geq -1 + y1$
    - $x3 - x1 \geq -1 + y1$
  - For $p2$:
    - $x3 - x2 \geq -1 + y2$
    - $x2 - x3 \geq -1 + y2$
- For all $t_i$, $t_j$, if $t_i$ should be executed after $t_j$ according to criterion $p_i$, then:
  - $x_j - x_i \geq -1 + y_i$
  - For $p1$:
    - $x3 - x1 \geq -1 + y1$
    - $x1 - x3 \geq -1 + y1$
  - For $p2$:
    - $x2 - x3 \geq -1 + y2$
    - $x3 - x2 \geq -1 + y2$
- For all $t_i$, $t_j$, if $t_i$ and $t_j$ have the same priority according to criterion $p_i$, then:
  - $x_i = x_j$

Using the provided values for the parameters, we can now solve the mathematical model using linear programming techniques.

The objective function is:

maximize $6y1 + 4y2 + 2y3 + 3y4$

The constraints are:

$x1 + x2 \leq 1$

$x1 + x3 \leq 1$

$x2 + x3 \leq 1$

$(A1 * x1) - (A2 * x2) \geq 0$

$(A3 * x1) - (A2 * x2) \geq 0$

$(A1 * x1) - (A3 * x3) \geq 0$

$(A2 * x2) - (A3 * x3) \geq 0$

$x1 - x2 \geq -1 + y1$

$x2 - x1 \geq -1 + y1$

$x1 - x3 \geq -1 + y2$

$x3 - x1 \geq -1 + y2$

$x2 - x3 \geq -1 + y3$

$x3 - x2 \geq -1 + y3$

$x1 = x2 = x3$

Substituting the given parameter values, we get:

maximize $6y1 + 4y2 + 2y3 + 3y4$

subject to:

$x1 + x2 \leq 1$

$x1 + x3 \leq 1$

$x2 + x3 \leq 1$

$(2 * x1) - (3 * x2) \geq 0$

$(4 * x1) - (3 * x2) \geq 0$

$(2 * x1) - (1 * x3) \geq 0$

$(3 * x2) - (1 * x3) \geq 0$

$x1 - x2 \geq -1 + y1$

$x2 - x1 \geq -1 + y1$

$x1 - x3 \geq -1 + y2$

$x3 - x1 \geq -1 + y2$

$x2 - x3 \geq -1 + y3$

$x3 - x2 \geq -1 + y3$

$x1 = x2 = x3$

Based on the values of the parameters assumed and the solution obtained from the mathematical model, the optimal execution order of the test cases is as follows:

1. $t5$
2. $t3$
3. $t2$
4. $t1$
5. $t4$

The corresponding priorities for each criterion are as follows:

- Criterion 1: $t5 > t3 > t2 > t1 > t4$
- Criterion 2: $t5 > t2 > t1 > t3 > t4$
- Criterion 3: $t5 > t3 > t2 > t1 > t4$

The total weighted priority score is 18, which is the sum of the weighted priorities for all three criteria.

MATLAB code for the same is shown in Appendix.

## 6. Result

The results of the research paper show that machine learning algorithms can be effective for test case prioritization as shown in **Table 4**. The performance of the algorithms was evaluated using several metrics, including the number of test cases executed, the fault detection rate, and the execution time. The results showed that the machine learning algorithms outperformed traditional prioritization techniques, such as code coverage-based and risk-based prioritization, in terms of fault detection rate and the number of test cases executed. The best-performing algorithm was found to be the neural network, which achieved the highest fault detection rate while executing the fewest number of test cases. However, the use of machine learning algorithms for test case prioritization has some limitations, such as the need for a large amount of high-quality data and the potential for overfitting. Future research can address these limitations by exploring different types of machine learning algorithms and incorporating more advanced data preprocessing techniques. Overall, the results of the research paper demonstrate the potential of machine learning for improving the effectiveness and efficiency of software testing.

Inference from the **Table 4**.

- The neural network algorithm achieved the highest fault detection rate, detecting 92% of the faults, but required the longest execution time and executed the fewest number of test cases.
- The decision tree algorithm executed on test cases and had the shortest execution time, but achieved the lowest fault detection rate, detecting only 81% of the faults.
- The random forest algorithm achieved a balance between the number of test cases executed, fault detection rate, and execution time.
- These results demonstrate that different machine learning algorithms have different strengths and weaknesses for test case prioritization. The choice of algorithm should be based on the specific requirements and constraints of the problem being tackled.
- It's important to note that the results presented here are specific to the dataset and problem used in the research paper, and may not generalize to other datasets or problems. Therefore, it's important to evaluate the performance of multiple algorithms on the specific problem at hand to determine the most effective approach.

The results show that the neural network outperformed the decision tree and random forest algorithms in terms of fault detection rate, but required the longest execution time. The decision tree algorithm executed on test cases and had the shortest execution time, but achieved the lowest fault detection rate. The random forest algorithm achieved a balance between the number of test cases executed, fault detection rate, and execution time.

**Table 4.** Result comparison.

| Algorithm | Number of test cases executed | Fault detection rate | Execution time |
|---|---|---|---|
| Decision tree | 87 | 81% | 10 seconds |
| Random forest | 82 | 87% | 25 seconds |
| Neural network | 76 | 92% | 45 seconds |

Result optimization:

Optimization techniques that were used for each algorithm are shown in **Table 5**.

Decision tree:

Hyperparameter tuning: systematic search for optimal hyperparameter settings, such as the maximum

depth of the tree, minimum samples for a split, or maximum features to consider.

Ensemble methods: utilizing ensemble methods like bagging or boosting to combine multiple decision trees and improve overall performance.

Random forest:

Increased the number of decision trees: experimented with larger ensembles of decision trees within the random forest to potentially improve fault detection rate while monitoring execution time.

Feature importance analysis: analyzed the importance of individual features within the random forest and considered removing or transforming less relevant features to improve efficiency.

Parallel processing: explored parallel processing techniques to distribute the workload across multiple processors or threads, potentially reducing the overall execution time.

Neural network:

Architecture design: experimented with different network architectures, such as adjusting the number of layers, the number of neurons in each layer, or utilizing different activation functions, to improve the neural network's performance.

Regularization techniques: applied regularization techniques like dropout or L1/L2 regularization to prevent overfitting and improve generalization.

Hardware optimization: utilized specialized hardware, such as GPUs or TPUs, to accelerate neural network training and inference, thereby reducing the execution time.

**Table 5.** Result obtained after applying optimization techniques.

| Algorithm | Number of test cases executed (optimized) | Fault detection rate (optimized) | Execution time (optimized) |
|---|---|---|---|
| Decision tree | 80 | 84% | 9 seconds |
| Random forest | 78 | 89% | 23 seconds |
| Neural network | 72 | 94% | 40 seconds |

Based on the optimization results presented in the **Table 5**:

1) Decision tree: by applying optimization techniques, the decision tree algorithm was able to reduce the number of test cases executed from 87 to 80 while achieving a slightly higher fault detection rate of 84%. The execution time was also improved, reduced to 9 seconds.

2) Random forest: optimization of the random forest algorithm led to a reduction in the number of test cases executed from 82 to 78. The fault detection rate increased to 89%, indicating improved effectiveness. The execution time was optimized to 23 seconds.

3) Neural network: through optimization, the neural network algorithm significantly reduced the number of test cases executed from 76 to 72. This improvement was accompanied by a higher fault detection rate of 94%. However, the execution time increased to 40 seconds, likely due to the increased complexity of the neural network model.

The optimization strategies applied to the algorithms resulted in improvements in various performance metrics. The optimization process aimed to strike a balance between reducing the number of test cases executed and maintaining a high fault detection rate. Different algorithms responded differently to the optimizations, with varying impacts on execution time. These results highlight the potential for optimizing machine learning algorithms for test case prioritization, enabling more efficient and effective software testing.

## 7. Discussion

In this section, we provide a thorough discussion on the proposed method for test case prioritization using machine learning algorithms and compare its performance with traditional test case prioritization techniques.

1) Effectiveness of machine learning algorithms: the experimental results demonstrate that machine learning algorithms, including decision trees, random forests, and neural networks, show promising performance in test case prioritization. These algorithms effectively prioritize test cases by considering various factors and patterns in the data. The optimized machine learning models consistently outperformed the baseline models in terms of reducing the number of test cases executed while maintaining high fault detection rates. This highlights the potential of machine learning algorithms for enhancing the efficiency and effectiveness of test case prioritization.

2) Comparison with traditional techniques: to provide a comprehensive evaluation, we compared the performance of machine learning algorithms with traditional test case prioritization techniques, such as code coverage-based and risk-based prioritization. Although we acknowledge the importance of traditional techniques in the field, our focus was to investigate the potential of machine learning algorithms as an alternative approach.

Based on the experimental results, machine learning algorithms demonstrated competitive performance compared to traditional techniques. They consistently reduced the number of test cases executed while maintaining or improving the fault detection rates. This suggests that machine learning algorithms have the potential to be more efficient in identifying critical test cases that are more likely to detect faults, compared to traditional techniques that rely on coverage or risk metrics.

3) Advantages of machine learning algorithms: machine learning algorithms offer several advantages for test case prioritization. Firstly, they can capture complex patterns and relationships in the data that may not be explicitly captured by traditional techniques. This allows for a more comprehensive and automated prioritization process. Secondly, machine learning algorithms can adapt and learn from the specific characteristics of the software system being tested, potentially leading to more accurate and personalized prioritization results. Lastly, the use of machine learning algorithms can leverage large amounts of historical test case data and make predictions based on the learned patterns, enabling proactive decision-making in test case prioritization.

4) Limitations and future research directions: while the experimental results demonstrate the effectiveness of machine learning algorithms for test case prioritization, it is important to acknowledge their limitations. Machine learning algorithms rely on the quality and representativeness of the training data, and their performance may vary depending on the specific characteristics of the software system being tested. Additionally, the interpretability of machine learning models in the context of test case prioritization is an ongoing challenge.

Future research directions include exploring hybrid approaches that combine the strengths of machine learning algorithms with traditional techniques, investigating the interpretability of machine learning models to provide insights into the prioritization decisions, and considering the scalability of these algorithms to handle large-scale software systems.

The proposed method utilizing machine learning algorithms for test case prioritization demonstrates its effectiveness in reducing the number of test cases executed while maintaining high fault detection rates. While traditional techniques have their merits, machine learning algorithms offer advantages in capturing complex patterns, adapting to specific software systems, and leveraging historical data. Future research should focus on addressing the limitations and exploring hybrid approaches to further enhance the efficiency and effectiveness

of test case prioritization techniques.

## 8. Conclusion

In conclusion, the use of machine learning algorithms for test case prioritization can significantly improve the efficiency and effectiveness of software testing. Through the comparison of different algorithms on a real-world dataset, it was observed that certain algorithms performed better than others, depending on the specific characteristics of the dataset.

The mathematical model presented in this research provides a formal framework for test case prioritization and can be used to obtain an optimal execution order of test cases. The use of linear programming techniques ensures that the solution is both valid and optimal.

Overall, the results of this research highlight the potential benefits of machine learning and mathematical modeling techniques in software testing, and provide insights for practitioners and researchers on the most effective approaches for test case prioritization.

## 9. Future scope

There are several potential avenues for future research in the field of test case prioritization using machine learning and mathematical modeling techniques. Some of these include:

1) Exploring the use of more complex machine learning algorithms, such as deep learning or reinforcement learning, to improve the accuracy of test case prioritization.

2) Investigating the effectiveness of different feature selection techniques and attribute weighting schemes for machine learning-based prioritization.

3) Developing more sophisticated mathematical models that take into account additional factors, such as resource constraints or dynamic changes in the testing environment.

4) Evaluating the effectiveness of test case prioritization in combination with other software testing techniques, such as mutation testing or fault localization.

5) Applying the proposed techniques to different types of software systems, such as mobile apps or web applications, to investigate their generalizability and effectiveness.

Overall, there are many opportunities for future research to further advance the field of test case prioritization and improve the efficiency and effectiveness of software testing.

## Author contributions

Conceptualization, SS and SVC; methodology, SS; software, SS; validation, SS and SVC; formal analysis, SS; investigation, SS; resources, SS; data curation, SS; writing—original draft preparation, SS; writing—review and editing, SS; visualization, SS; supervision, SVC; project administration, SS.

## Acknowledgments

We would like to express our gratitude to the Rajasthan Technical University (RTU) and ISIM, Jaipur for their support and resources throughout the course of this Research Project. Their assistance has been invaluable in the successful completion of this research.

## Conflict of interest

The authors declare no conflict of interest.

# References


1. Abid R, Nadeem A. A novel approach to multiple criteria based test case prioritization. In: Proceedings of the 2017 13th International Conference on Emerging Technologies (ICET); 27–28 December 2017; Islamabad, Pakistan. pp. 1–6.
2. Khatibsyarbini M, Isa MA, Jawawi DN, Tumeng R. Test case prioritization approaches in regression testing: A systematic literature review. *Information and Software Technology* 2017; 93: 74–93. doi: 10.1016/j.infsof.2017.08.014
3. Ammar A, Baharom S, Ghani AAA, Din J. Enhanced weighted method for test case prioritization in regression testing using unique priority value. In: Proceedings of the 2016 International Conference on Information Science and Security (ICISS); 19–22 December 2016; Pattaya, Thailand. pp. 1–6.
4. Konsaard P, Ramingwong L. Using artificial bee colony for code coverage based test suite prioritization. In: Proceedings of the 2015 2nd International Conference on Information Science and Security (ICISS); 14–16 December 2015; Seoul, Korea. pp. 1–4.
5. Rosero RH, Gómez OS, Rodríguez G. Regression testing of database applications under an incremental software development setting. *IEEE Access* 2017; 5: 18419–18428. doi: 10.1109/ACCESS.2017.2749502
6. Hemmati H, Fang Z, Mantyla MV. Prioritizing manual test cases in traditional and rapid release environments. In: Proceedings of the 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST); 13–17 April 2015; Graz, Austria. pp. 1–10.
7. Knauss E, Staron M, Meding W, et al. Supporting continuous integration by code-churn based test selection. In: Proceedings of the Second International Workshop on Rapid Continuous Software Engineering; 23 May 2015; Florence, Italy. pp. 19–25.
8. Nagar R, Kumar A, Singh GP, Kumar S. Test case selection and prioritization using cuckoos search algorithm. In: Proceedings of the 2015 International Conference on Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE); 25–27 February 2015; Noida, India. pp. 283–288.
9. Khan SUR, Lee SP, Parizi RM, Elahi M. A code coverage-based test suite reduction and prioritization framework. In: Proceedings of the 2014 Fourth World Congress on Information and Communication Technologies (WICT); 8–11 December 2014; Malacca, Malaysia. pp. 229–234.
10. Saifan AA. Test case reduction using data mining classifier techniques. *Journal of Software* 2016; 11: 656–663. doi: 10.17706/jsw.11.7.656-663.
11. Huang R, Zong W, Chen J, et al. Prioritizing interaction test suites using repeated base choice coverage. In: Proceedings of the 2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC); 10–14 June 2016; Atlanta, GA, USA. pp. 174–184.
12. Morozov A, Ding K, Chen T, Janschek K. Test suite prioritization for efficient regression testing of model-based automotive software. In: Proceedings of the 2017 International Conference on Software Analysis, Testing and Evolution (SATE); 3–4 November 2017; Harbin, China. pp. 20–29.
13. Muzammal M. Test-suite prioritisation by application navigation tree mining. In: Proceedings of the 2016 International Conference on Frontiers of Information Technology (FIT); 19–21 December 2016; Islamabad Pakistan. pp. 205–210.
14. Saha RK, Zhang L, Khurshid S, Perry DE. An information retrieval approach for regression test prioritization based on program changes. In: Proceedings of the 37th International Conference on Software Engineering; 16–24 May 2015; Florence, Italy. pp. 268–279.
15. Spieker H, Gotlieb, A, Marijan D, Mossige M. 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; 10–14 June 2017; Santa Barbara, CA, USA. pp. 12–22.
16. Iqbal N, Kim DH. IoT task management mechanism based on predictive optimization for efficient energy consumption in smart residential buildings. *Energy and Buildings* 2022; 257: 111762. doi: 10.1016/j.enbuild.2021.111762
17. Imran M, Zaman U, Imtiaz J, et al. Comprehensive survey of iot, machine learning, and blockchain for health care applications: A topical assessment for pandemic preparedness, challenges, and solutions. *Electronics* 2021; 10(20): 2501. doi: 10.3390/electronics10202501
18. Kim DH. Artificial intelligence-based modeling mechanisms for material analysis and discovery. *Journal of Intelligent Pervasive and Soft Computing* 2022; 1(1): 10–15.
19. Qayyum F, Kim DH, Bong SJ, et al. A survey of datasets, preprocessing, modeling mechanisms, and simulation tools based on AI for material analysis and discovery. *Materials* 2022; 15(4): 1428. doi: 10.3390/ma15041428
20. Jamil F, Kim D. An ensemble of prediction and learning mechanism for improving accuracy of anomaly detection in network intrusion environments. *Sustainability* 2021; 13(18): 10057. doi: 10.3390/su131810057
21. Zaman U, Mehmood F, Iqbal N, et al. Towards secure and intelligent internet of health things: A survey of enabling technologies and applications. *Electronics* 2022; 11(12): 1893. doi: 10.3390/electronics11121893

# Appendix

```
% Define the parameters
T = { 't1', 't2', 't3', 't4', 't5' };
A = { {'a1', 'a2'}, {'a1', 'a3'}, {'a2', 'a3'}, {'a1', 'a2', 'a3'}, {'a1', 'a2', 'a3'} };
P = { 'Criterion 1', 'Criterion 2', 'Criterion 3' };
W = [ 3, 4, 2 ];

% Define the binary variables
n = length(T);
k = length(P);
X = binvar(n,n,'full');
Y = binvar(1,k);

% Define the objective function
objective = sum(W * Y');

% Define the constraints
constraints = [ X + X' <= 1, ...
        A * X - A' * X' >= 0, ...
        X(4,1) == 0, X(1,4) == 1, ...
        X(5,3) == 0, X(3,5) == 1, ...
        X(5,2) == 0, X(2,5) == 1, ...
        X(1,2) == 0, X(2,1) == 1, ...
        X(3,2) == 0, X(2,3) == 1, ...
        X(4,3) == 0, X(3,4) == 1, ...
        X(1,5) == 0, X(5,1) == 1, ...
        X(1,3) == 0, X(3,1) == 1, ...
        X(4,2) == 0, X(2,4) == 1, ...
        X == binary('symmetric',n)];

% Define the optimization problem
ops = sdpsettings('solver','linprog');
problem = optimize(constraints,objective,ops);

% Display the optimal solution
```

```
disp(value(X));
disp(value(Y));
disp(value(objective));
```