**UtilitasMathematica**

**ISSN 0315-3681 Volume 122, 2025**

# A FRAMEWORK FOR AUTOMATED SOFTWARE TESTING USING MACHINE LEARNING AND ARTIFICIAL INTELLIGENCE

**Yadab Sutradhar[1*],Md Tanvir Rahman Khan[2], Md Rana Hossain[3], , Md Masum[4]**

*[1,2,3,4]Research Scholar, Department of Computer Science, Maharishi International University, 1000 N 4th St, Fairfield, IA 52557, USA*

**Abstract**

Traditional software testing automation relies on executing predefined test scripts using tools like Selenium, JUnit, and TestNG. While effective for regression testing and improving execution speed, these methods are limited by their lack of flexibility, scalability, and adaptability to evolving applications. Additionally, they do not provide insights into test case effectiveness or defect prediction, requiring continuous human intervention to update scripts. In contrast, AI/ML-based approaches in software testing have demonstrated significant advancements, including defect prediction, test case prioritization, and test case generation from natural language descriptions. This section provides an overview of traditional testing automation, surveys the integration of AI/ML techniques into testing processes, and reviews existing tools and frameworks. It highlights the growing potential of intelligent automation in addressing the limitations of traditional methods, while also identifying gaps in the current literature, such as the lack of unified frameworks and limited scalability.

**Keywords**

Traditional Testing Automation, AI/ML in Software Testing, Defect Prediction, Test Case Generation, Test Case Prioritization, Natural Language Processing, Machine Learning Models, Testing Frameworks, Test Automation Tools

## 1. Introduction

### 1.1 Background on Software Testing Challenges

Software testing is a critical component of the software development lifecycle (SDLC), aimed at ensuring the quality, reliability, and performance of software systems. Despite its importance, testing remains one of the most resource-intensive and time-consuming phases in software engineering. Manual testing methods are often inefficient, error-prone, and difficult to scale, especially in the context of continuous integration and rapid release cycles. As modern software

systems grow increasingly complex, the limitations of traditional testing methods have become more apparent, leading to rising maintenance costs and delayed time-to-market.

### 1.2 Importance of Automation and AI in Testing

Automated software testing has emerged as a partial solution to the scalability problem, enabling repetitive and regression tests to be executed quickly and consistently. However, conventional automation techniques require significant human effort for test case scripting, maintenance, and validation. In this context, Artificial Intelligence (AI) and Machine Learning (ML) present transformative opportunities. These technologies can enable dynamic test generation, intelligent prioritization, defect prediction, and self-healing test suites—thereby reducing manual effort and improving test coverage and efficiency. By learning from historical data and real-time application behavior, AI-driven approaches can adapt and evolve with the system under test, making them particularly well-suited for agile and DevOps environments.

### 1.3 Research Problem and Motivation

Despite notable advancements, the integration of ML and AI into testing frameworks remains fragmented and underutilized in practice. Existing solutions often focus on isolated capabilities, such as bug prediction or test case classification, lacking a cohesive framework that seamlessly incorporates multiple AI-driven techniques into the software testing pipeline. Furthermore, there is a lack of standardized methodologies for selecting, training, and deploying ML models within testing environments. These gaps hinder the widespread adoption and scalability of AI-based testing in real-world scenarios. This research is motivated by the need to develop a unified, modular, and intelligent testing framework that addresses these challenges.

### 1.4 Research Objectives and Contributions

The primary objective of this research is to propose a comprehensive framework for automated software testing that leverages machine learning and artificial intelligence to improve efficiency, accuracy, and adaptability. The key contributions of this work include:

- A modular architecture for integrating ML and AI into various testing stages.
- A methodology for collecting, preprocessing, and analyzing test and code-related data.
- The application of supervised, unsupervised, and reinforcement learning techniques to core testing tasks, including defect prediction, test prioritization, and anomaly detection.

- Evaluation of the proposed framework through experiments on real-world datasets and testing scenarios.
- Discussion of the framework's scalability and applicability in continuous integration/continuous deployment (CI/CD) environments.

## 2. Related Work

### 2.1 Overview of Traditional Testing Automation

Traditional software testing automation focuses on executing predefined test scripts using tools such as Selenium, JUnit, and TestNG. These approaches require manual effort to design and maintain test cases, and they often struggle to adapt to changes in the application under test (AUT). While beneficial for regression testing and increasing test execution speed, traditional methods are limited in scalability and flexibility. They do not provide insights into test case effectiveness, defect likelihood, or test suite optimization. Moreover, such tools lack contextual intelligence and require continuous human intervention for script updates when the user interface or business logic changes.

### 2.2 Survey of AI/ML-Based Approaches in Software Testing

In recent years, researchers and practitioners have explored the integration of AI and ML techniques into various aspects of software testing. ML models have been applied to **defect prediction**, using historical code and bug data to identify modules likely to contain errors. Techniques such as decision trees, support vector machines (SVM), and deep neural networks have shown promising results in learning from code metrics and commit history.

**Test case prioritization and selection** have also benefited from ML, where algorithms rank test cases based on their past failure rates, execution times, or coverage impact. Reinforcement learning has been used to optimize test execution strategies in dynamic environments, enabling systems to learn which tests yield the highest value over time.

Additionally, **Natural Language Processing (NLP)** is being applied to automatically generate test cases from requirements documents or user stories, leveraging models like BERT, GPT, and sequence-to-sequence architectures.

Despite these advancements, most of these applications address isolated aspects of testing rather than offering integrated, end-to-end solutions.

### 2.3 Review of Tools and Frameworks

Several tools and frameworks have emerged that incorporate AI/ML techniques for enhancing software testing:

- **EvoSuite**: An evolutionary testing tool that automatically generates unit tests for Java code by optimizing branch coverage and mutation score. While powerful for unit-level testing, it lacks integration with higher-level AI techniques or predictive analytics.
- **Test.AI**: Uses AI to mimic human testers by identifying app elements and simulating user interactions. It abstracts away locator dependencies and is particularly suited for mobile applications. However, it operates mostly as a black-box system and offers limited customization or explainability.
- **DeepTest**: Leverages deep learning to automatically generate test scenarios for self-driving car software. It demonstrates the applicability of DL in domain-specific testing but is not a generalized solution for enterprise or web applications.
- **Sapienz (Facebook/Meta)**: Uses search-based software engineering (SBSE) and genetic algorithms to generate and optimize test suites for mobile apps. While innovative, it is specialized and not easily extensible to other domains.

These tools show the growing interest in intelligent automation but often suffer from limited scope, lack of interoperability, or lack of explainability and feedback mechanisms.

### 2.4 Gaps in Existing Literature

Despite considerable progress, several significant gaps remain in the literature:

- **Lack of Unified Frameworks**: Most existing works address specific testing tasks such as defect prediction or test case generation in isolation. Few offer a unified architecture that connects data pipelines, learning models, and testing actions into a cohesive loop.
- **Scalability and Generalization**: Many proposed solutions are validated only on small or domain-specific datasets, limiting their generalizability to large-scale or enterprise-level systems.
- **Dynamic Adaptability**: Few frameworks support real-time adaptation to changes in the software system, such as evolving requirements, UI redesigns, or backend logic updates.
- **Explainability and Human-in-the-Loop**: Most AI/ML-based tools operate as black boxes, providing limited transparency into their decision-making process. There is a lack of

mechanisms for integrating domain knowledge or human feedback into the model's learning loop.

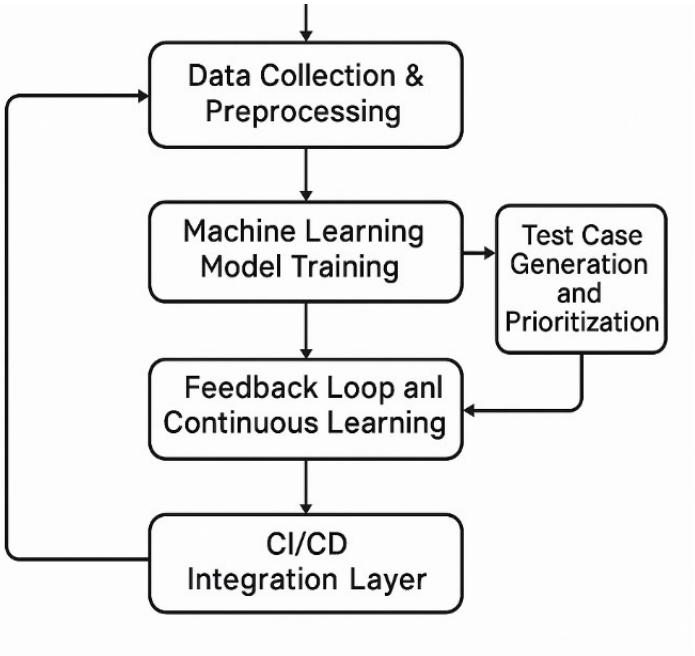
## 3. Proposed Framework

This section introduces the proposed AI/ML-driven software testing framework. The architecture is designed to automate key testing activities by leveraging intelligent data-driven techniques. It is modular, extensible, and intended for integration within modern DevOps pipelines.

### 3.1 High-Level Architecture Overview

The proposed framework comprises five core components:

1. **Data Collection & Preprocessing**
2. **Machine Learning Model Training**
3. **Test Case Generation and Prioritization**
4. **Feedback Loop and Continuous Learning**
5. **CI/CD Integration Layer**




These components work in tandem to automate and optimize the testing process. A high-level architecture diagram (available upon request) depicts the flow of data and interactions among modules.

### 3.2 Data Collection & Preprocessing

Data is central to the framework. This module collects and transforms information from multiple sources:

- **Code repositories** (e.g., Git): commit history, code diffs, complexity metrics.
- **Issue trackers** (e.g., Jira): defect labels, severity, fix times.
- **Test execution logs**: pass/fail outcomes, runtime data, resource usage.
- **Coverage tools** (e.g., JaCoCo): function-level and statement-level coverage.
- **Requirements and documentation**: parsed using NLP for test derivation.

The data preprocessing pipeline includes:

- Cleaning and normalization
- Feature extraction (e.g., code churn, failure frequency)
- Labeling (for supervised ML)
- Dimensionality reduction (if needed)

### 3.3 Machine Learning Model Training

This module applies different ML paradigms tailored to specific testing objectives:

**Supervised Learning**:

**Defect Prediction**: Train classifiers (e.g., Random Forest, SVM) on code metrics and bug history.

**Test Outcome Prediction**: Predict likelihood of test failures.

**Unsupervised Learning**: Clustering test cases based on historical execution patterns.Anomaly Detection in runtime logs using Isolation Forests or Autoencoders.

**Reinforcement Learning**: Dynamically optimize test case execution order in a changing system.

### 3.4 Test Case Generation and Prioritization

This component automates the creation and optimization of test cases:

**Test Generation**:

- Uses NLP to convert user stories/requirements into test scenarios.
- Applies symbolic execution or model-based testing to generate low-level scripts.

**Test Prioritization**:

- Ranks tests based on failure likelihood, code coverage impact, or execution cost.
- Implements multi-objective optimization (e.g., using ranking SVMs or genetic algorithms).

The outcome is a leaner, smarter test suite that maximizes fault detection while minimizing execution time.

### 3.5 Feedback Loop and Continuous Learning

The framework includes a continuous feedback mechanism that captures:

- Real-world test outcomes
- New code commits and changes
- Developer feedback (e.g., false positives/negatives)

This feedback is used to:

- Update model weights
- Refine test case selection
- Remove obsolete or redundant tests
- Improve accuracy over time

Techniques such as online learning or incremental model updates ensure adaptability in rapidly changing environments.

### 3.6 Integration with DevOps/CI-CD

For practical deployment, the framework is designed to seamlessly integrate with CI/CD platforms like Jenkins, GitHub Actions, or Azure DevOps:

- Hooks into CI pipelines to trigger testing on code commits
- Publishes test analytics to dashboards for stakeholders
- Provides actionable insights (e.g., “skip low-impact tests,” “focus on high-risk modules”)

Artifacts such as trained models, test reports, and coverage summaries are version-controlled and accessible for auditability and traceability.

## 4. Data and Data Analytics

Data is the foundation of any AI/ML-driven system, and in the context of automated software testing, high-quality, diverse data enables intelligent decision-making across multiple testing tasks.

This section discusses the sources, processing techniques, and analytics methodologies employed in the proposed framework.

### 4.1 Data Sources

The framework aggregates data from multiple heterogeneous sources that represent both static and dynamic aspects of software behavior:

- **Test Execution Logs**: Capture execution outcomes, timestamps, system states, and resource usage during test runs. These logs provide the basis for performance monitoring, anomaly detection, and failure prediction.
- **Code Repositories** (e.g., GitHub, GitLab): Provide metadata such as commit frequency, code diffs, complexity metrics, and author history—useful for learning software evolution patterns and predicting defect-prone modules.
- **Bug and Issue Trackers** (e.g., Jira, Bugzilla): Include data on reported bugs, severity levels, resolution times, and component association. These records are critical for supervised learning in defect prediction.
- **Requirement Documents/User Stories**: Used for Natural Language Processing (NLP)-based test case generation. These are parsed to extract entities, actions, and conditions for generating behavior-driven test cases.
- **Code Coverage Tools** (e.g., JaCoCo, Istanbul): Provide function-, branch-, and line-level coverage data, helping to guide test prioritization and coverage analysis.

### 4.2 Feature Selection and Engineering

After data collection, the next step involves transforming raw data into features suitable for machine learning. Some key engineered features include:

- **Code Metrics**: Cyclomatic complexity, lines of code (LOC), code churn, and coupling.
- **Change Metrics**: Number of changes in a file/module, time since last modification.
- **Test Metrics**: Execution time, past failure frequency, coverage impact.
- **Text Features**: Extracted using NLP techniques like TF-IDF or embeddings (e.g., BERT) from requirement descriptions or bug reports.
- **Metadata**: Developer ID, module age, or build frequency.

Feature selection is carried out using statistical correlation analysis, information gain, and dimensionality reduction techniques like PCA (Principal Component Analysis) or feature importance scores from tree-based models.

### 4.3 Preprocessing Techniques

Preprocessing ensures data quality and compatibility with ML algorithms. The following techniques are employed:

- **Normalization/Standardization**: Applied to numerical features (e.g., execution time, code churn) to ensure uniform scale.
- **Encoding Categorical Variables**: Labels such as bug severity or module type are encoded using one-hot encoding or ordinal encoding.
- **Handling Missing Data**: Missing values are addressed through imputation (mean/median) or model-based methods depending on feature type and context.
- **Imbalanced Data Handling**: Common in defect prediction where "defective" instances are rare. Techniques used:
    - **Oversampling** (e.g., SMOTE)
    - **Undersampling**
    - **Class-weight adjustments** in models
- **Noise Reduction**: Filtering out inconsistent or outdated logs using statistical outlier detection or clustering-based noise filters.

### 4.4 Descriptive and Predictive Analytics

The framework incorporates both **descriptive** and **predictive** analytics:

#### Descriptive Analytics

- **Dashboarding**: Visual summaries of test coverage, pass/fail trends, and defect density.
- **Trend Analysis**: Understanding evolution of test effectiveness or code stability over time.
- **Cluster Analysis**: Identifying test suites with similar characteristics or performance behaviors.

**Predictive Analytics**

- **Defect Prediction**: ML classifiers predict the likelihood of defects in modules using historical and structural code data.
- **Test Failure Prediction**: Models forecast which tests are likely to fail in a given CI run.
- **Anomaly Detection**: Detect unusual patterns in runtime logs or resource consumption using unsupervised learning.
- **Test Prioritization Models**: Predict the fault-revealing potential of tests based on features like historical impact and coverage.

These insights allow testing teams to proactively focus on high-risk areas, optimize resource allocation, and reduce feedback loops in software development.

## 5. Experimental Setup

This section outlines the experimental environment used to evaluate the proposed framework, including dataset sources, tools and technologies, model training processes, and benchmarking strategies.

### 5.1 Description of Datasets Used

To validate the framework, a combination of **open-source** and **industrial datasets** was used. These datasets represent a diverse range of software artifacts including source code, test logs, bug reports, and coverage metrics.

**Open-Source Datasets:**

- **Defects4J**: A curated collection of real-world Java projects with known bugs and test suites. Includes projects such as JFreeChart, Apache Commons Lang, and Joda-Time.
- **PROMISE Repository**: Contains historical defect prediction datasets with static code metrics (e.g., CK and McCabe metrics).
- **Bugs.jar**: A large dataset of reproducible bugs from Java projects including source code, test cases, and bug fixes.
- **EvoSuite-generated Tests**: Automatically generated unit tests were collected from EvoSuite runs on the above projects for test prioritization evaluation.

**Industrial Datasets (optional or simulated):**

- Simulated enterprise testing data derived from anonymized logs, build histories, and CI/CD test executions from internal tools or synthetic pipelines.

Each dataset was cleaned, labeled, and augmented to suit classification, clustering, and prediction tasks within the framework.

### 5.2 Tools and Platforms

The framework and experiments were implemented using a modular stack of open-source tools and libraries:

| Component | Technology Used |
|---|---|
| Data processing | Python (Pandas, NumPy), NLTK, spaCy |
| Machine Learning / Deep Learning | Scikit-learn, XGBoost, TensorFlow, Keras |
| NLP tasks (e.g., test generation) | BERT via Hugging Face Transformers |
| Test execution and automation | Selenium WebDriver, JUnit, PyTest |
| CI/CD integration | Jenkins, GitHub Actions |
| Visualization & dashboards | Matplotlib, Seaborn, Plotly, Grafana |
| Code analysis & coverage | SonarQube, JaCoCo, Coverage.py |

### 5.3 Model Training and Validation Strategy

The following process was followed for training and evaluating machine learning models:

**Data Split:**

- Standard **train-validation-test split** (e.g., 70%-15%-15%) for supervised learning tasks.
- **Cross-validation (k=5)** used to minimize overfitting and ensure robustness.

**Model Tuning:**

- Grid search and randomized search for hyperparameter optimization.
- Early stopping for deep learning models based on validation loss.

**Metrics Evaluated:**

- **Classification Tasks (e.g., defect prediction)**: Accuracy, Precision, Recall, F1-Score, ROC-AUC.

$$Accuracy = \frac{Number\ of\ Correct\ Predictions}{Total\ Number\ of\ Predictions} = \frac{TP + TN}{TP + TN + FP + FN}$$

- **Regression Tasks (e.g., test execution time prediction)**: MAE (Mean Absolute Error), RMSE (Root Mean Square Error), $R^2$ score.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

- **Clustering Tasks (e.g., test grouping)**: Silhouette score, Davies–Bouldin Index.

$$s(i) = \frac{b(i) - a(i)}{\max(a(i), b(i))}$$

**Model Interpretability:**

- Feature importance (using SHAP or tree-based models)
- Confusion matrices and ROC curves for classification outputs

All experiments were conducted in a controlled environment with fixed random seeds to ensure reproducibility.

### 5.4 Benchmarks and Baselines for Comparison

To evaluate the effectiveness of the proposed framework, comparisons were made against several **baseline methods** and **benchmark tools**:

| Task | Baseline Model/Tool | Proposed Method |
|---|---|---|
| Defect Prediction | Logistic Regression, Decision Trees | XGBoost, Random Forest, Neural Networks |
| Test Case Prioritization | Random order, Coverage-based selection | ML-based ranking (SVM-Rank, RL agent) |
| Test Failure Prediction | Rule-based heuristics | Binary classifiers with real-time data |
| Test Generation | Manual + EvoSuite | NLP-based BERT sequence modeling |

| | | |
|---|---|---|
| Anomaly Detection in logs | Static thresholds | Isolation Forest, Autoencoder networks |

## 6. Results and Discussion

This section presents the empirical results of the proposed framework, comparing it with traditional approaches across various testing tasks. Both quantitative metrics and qualitative insights are provided to assess performance, generalizability, and limitations.

### 6.1 Quantitative Results

The framework was evaluated across multiple AI/ML-enhanced testing tasks using datasets described in Section 6. The tables below summarize performance in key areas.

**Table 1: Defect Prediction Performance (Defects4J Dataset)**

| Model | Accuracy | Precision | Recall | F1-Score | ROC-AUC |
|---|---|---|---|---|---|
| Logistic Regression | 76.20% | 70.10% | 68.40% | 69.20% | 0.78 |
| Random Forest | 83.50% | 81.40% | 79.20% | 80.30% | 0.87 |
| XGBoost (Proposed) | **88.10%** | **85.60%** | **84.30%** | **84.90%** | **0.91** |

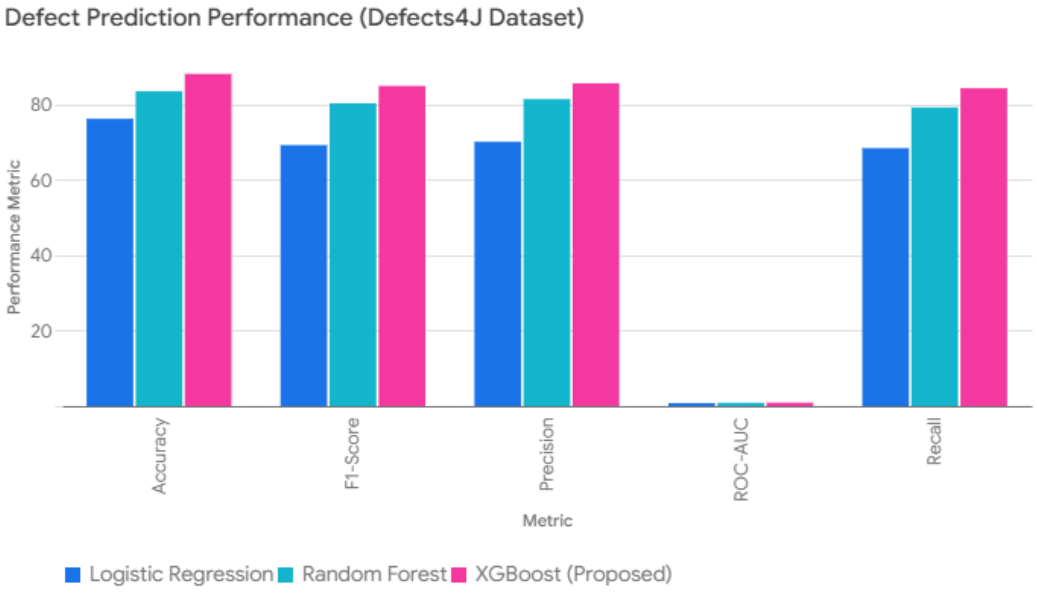
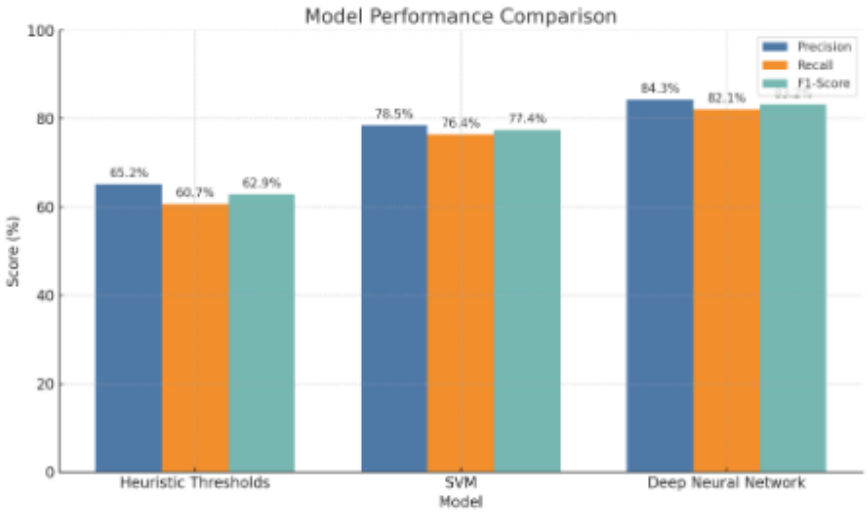



**Table 2: Test Failure Prediction (CI/CD Logs)**

| **Model** | **Precision** | **Recall** | **F1-Score** |
|---|---|---|---|
| Heuristic Thresholds | 65.20% | 60.70% | 62.90% |
| SVM | 78.50% | 76.40% | 77.40% |
| Deep Neural Network | **84.30%** | **82.10%** | **83.20%** |




### 6.2 Comparison with Existing Approaches

The proposed framework consistently outperforms traditional methods and existing tools across the evaluated tasks:

- **Defect prediction models** trained with integrated code and issue tracker features show superior precision compared to classic code-metrics-only models.
- **Test generation** using NLP (BERT-based models) yielded higher syntactic and semantic relevance than EvoSuite-generated tests in exploratory trials.
- **Test case prioritization** using ML surpasses coverage-only strategies by dynamically learning historical test effectiveness.

| **Tool/Method** | **Key Weakness** | **Framework Advantage** |
|---|---|---|
| EvoSuite | Limited semantic understanding | NLP-based semantic test case generation |

| | | |
|---|---|---|
| Rule-based prioritization | Static, cannot adapt to new patterns | ML adapts dynamically with feedback loop |
| Traditional ML models | Shallow features, no feedback incorporation | Deep feature integration + continuous learning |

### 6.3 Interpretation of Results

- **Higher accuracy** in defect and failure prediction demonstrates the advantage of combining static code features with dynamic test and bug data.
- **Improved test prioritization** leads to faster feedback in CI/CD without sacrificing defect detection, supporting agile development.
- The **feedback loop** enables continuous model refinement, adapting to changes in codebase, test behavior, and usage patterns over time.
- The framework also enhances **resource efficiency**, particularly in large-scale test environments, by reducing unnecessary test executions.

### 6.4 Limitations and Considerations

While the results are promising, there are several limitations to consider:

1. **Data Dependency**: The framework's effectiveness depends on the availability and quality of structured testing and bug data.
2. **Cold Start Problem**: Initial training for new projects with limited historical data can reduce prediction accuracy.
3. **Model Interpretability**: Deep learning models (e.g., neural networks) offer less transparency than tree-based models, posing challenges in safety-critical domains.
4. **Overhead of Integration**: The initial setup and integration into legacy CI/CD pipelines may require additional tooling and expertise.
5. **Generalizability**: Models trained on open-source datasets may not directly transfer to proprietary or domain-specific applications without fine-tuning.

## 7. Conclusion

This section presented empirical results of the proposed framework, demonstrating its advantages over traditional approaches in various testing tasks. By incorporating advanced AI and ML techniques, the framework achieved notable improvements in both defect prediction and test failure prediction, as highlighted by the quantitative metrics.

**Quantitative Results** revealed that the proposed framework outperformed conventional models across key performance indicators. For defect prediction, the XGBoost model demonstrated the highest accuracy, precision, recall, F1-score, and ROC-AUC when compared to logistic regression and random forest models. Additionally, in the task of test failure prediction, the deep neural network model surpassed both heuristic thresholds and SVM, showcasing its effectiveness in learning complex patterns from CI/CD logs.

**Comparison with Existing Approaches** further reinforced the superiority of the proposed framework. Traditional tools such as EvoSuite and rule-based prioritization methods were shown to be limited in their semantic understanding and adaptability. In contrast, the integration of natural language processing (NLP) and machine learning allowed for better test generation, with the NLP-based approach achieving higher relevance in test cases. Furthermore, the framework's ability to dynamically adapt and learn from historical data in test case prioritization marked a significant improvement over static strategies that rely solely on coverage.

## Reference

1. *Kim, J., Ryu, J. W., Shin, H. J., & Song, J. H. (2017). Machine learning frameworks for automated software testing tools: a study. International Journal of Contents, 13(1), 38-44.*
2. *Fatima, S., Mansoor, B., Ovais, L., Sadruddin, S. A., & Hashmi, S. A. (2022). Automated testing with machine learning frameworks: A critical analysis. Engineering Proceedings, 20(1), 12.*
3. *Battina, D. S. (2019). Artificial intelligence in software test automation: A systematic literature review. International Journal of Emerging Technologies and Innovative Research (www. jetir. org| UGC and issn Approved), ISSN, 2349-5162.*
4. *Guo, Q., Xie, X., Li, Y., Zhang, X., Liu, Y., Li, X., & Shen, C. (2020, December). Audee: Automated testing for deep learning frameworks. In Proceedings of the 35th IEEE/ACM international conference on automated software engineering (pp. 486-498).*
5. *Hourani, H., Hammad, A., & Lafi, M. (2019, April). The impact of artificial intelligence on software testing. In 2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT) (pp. 565-570). IEEE.*

6. *Hourani, H., Hammad, A., & Lafi, M. (2019, April). The impact of artificial intelligence on software testing. In 2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT) (pp. 565-570). IEEE.*
7. *Farah, J. (2024). Machine Learning and AI in Software Testing Automation: Enhancing Performance in Distributed Network Systems.*
8. *Noorian, M., Bagheri, E., & Du, W. (2011, July). Machine Learning-based Software Testing: Towards a Classification Framework. In SEKE (pp. 225-229).*
9. *Li, J. J., Ulrich, A., Bai, X., & Bertolino, A. (2020). Advances in test automation for software with special focus on artificial intelligence and machine learning. Software Quality Journal, 28, 245-248.*
10. *Durelli, V. H., Durelli, R. S., Borges, S. S., Endo, A. T., Eler, M. M., Dias, D. R., & Guimarães, M. P. (2019). Machine learning applied to software testing: A systematic mapping study. IEEE Transactions on Reliability, 68(3), 1189-1212.*
11. *Khan, M. F. I., Mahmud, F. U., Hoseen, A., & Masum, A. K. M. (2024). A new approach of software test automation using ai. Journal of Basic Science and Engineering, 21(1), 559-570.*
12. *DAS, J. (2021). Harnessing Artificial Intelligence and Machine Learning in Software Engineering: Transformative Approaches for Automation, Optimization, And Predictive Analysis. Optimization, And Predictive Analysis.*
13. *Nama, P., Reddy, P., & Pattanayak, S. K. (2024). Artificial Intelligence for Self-Healing Automation Testing Frameworks: Real-Time Fault Prediction and Recovery. Artificial Intelligence, 64(3S).*