

Machine Learning Models for Intelligent Test Case Selection

Yousof Darwish, Mohammed Al-Refai(✉), and Ahmed Alzubi

Computer Science Department, Faculty of Computer and Information Technology, Jordan University of Science and Technology, P.O. Box 3030, Irbid 22110, Jordan
mnalrefai@just.edu.jo

Abstract. Regression testing is running test cases after every code change, ensuring that there are no intended breaks in the changes. This study explores the application of machine learning for intelligent test case selection, aiming to reduce testing effort while maintaining test effectiveness. Using a dataset of test case features extracted from commits within the Apache Commons Collections project, we investigated the performance of four classification models: Logistic Regression, Random Forest, Support Vector Classifier (SVC), and Gradient Boosting. Test case selection labels were generated using Ekstazi, a regression test selection technique, to indicate whether a test case would be selected or not. The models were trained and evaluated using a group-based split strategy based on commit hashes to ensure realistic performance estimates that match what is expected in production. Evaluation metrics included accuracy, precision, recall, F1 score, AUC-ROC, and Regression Test Selection (RTS) metrics like safety violation, precision violation, and test suite reduction. The results indicate that Gradient Boosting achieved a favorable balance between test suite reduction and risk mitigation, offering a promising approach for automated test case selection within the context of regression testing. This work highlights the potential of machine learning to replicate and potentially outperform established techniques such as Ekstazi to optimize software testing processes and improve overall development efficiency.

Keywords: Regression test selection · Java software systems · Ekstazi · Static and dynamic analysis · Machine learning

1 Introduction

Regression testing refers to the process of executing test cases following each code modification to verify that no unintentional bugs or failures have been introduced, the number of test cases is generally high, and running them all on each change is costly. Therefore, regression test selection (RTS) optimize regression tests by selection test cases that got affected by code change. Big numbers of research are being dedicated to reduce the cost of regression testing using the approach of RTS [1,2]. To address this challenge, regression test selection

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2026
O. Gervasi et al. (Eds.): ICCSA 2025, LNCS 15898, pp. 377–387, 2026.
https://doi.org/10.1007/978-3-031-97657-5_23

(RTS) tools where developed to reduce the effort of selecting all test cases, and choose the affected test cases only, Among these tools, Ekstazi [3] has shown exceptional results by reducing the number of test cases while ensuring that the selected test cases still effectively evaluate the changes where done. However, these traditional RTS approaches and tools may not always work well in the complex and dynamic nature of the modern systems. In recent years, machine learning has proven itself as an amazing tool for optimizing software testing process. By using historical data, machine learning models can learn patterns and predict the relevance of test cases based on the code changes, offering more flexible approaches to test selection. While the use of machine learning is not new, there still limited research on it's practical application and comparison with established RTS techniques like Ekstazi. This study shows the potential of applying machine learning to the intelligent test case selection within the domain of regression testing. Specifically, we investigate four widely used classification models (Logistic Regression, Random Forest, Support Vector Classifier (SVC), and Gradient Boosting) to demonstrate their effectiveness in the RTS problem. We use a dataset of test case features derived from commits in the apache commons collections project, and evaluate the models using different metrics, including accuracy, precision, recall, and domain-specific metrics like safety violation and test suite reduction.

2 Related Work

A good number of studies have explored techniques to improve software testing, particularly in the domains of regression testing, automation, and the application of machine learning and deep learning techniques. Below, we summarize relevant works that inform and contextualize our approach.

Larsson and Höstklint (2021) – Dynamic Test Case Selection using Machine Learning

In their master's thesis, Larsson and Höstklint [4] propose a machine learning-based model for dynamic test case selection within regression testing. The primary goal is to reduce the number of test cases executed after any code change, without significantly impacting fault detection. They build a supervised learning model using features such as test execution history, code coverage data, and file modification history. Their approach shows that machine learning model can be used to select subsets of test cases and achieve comparable effectiveness to running the full tests, with considerable time savings. This work contributes a practical application of ML to regression testing and demonstrate the importance of feature engineering in learning-based test selection.

Yoo and Harman (2012) – Regression Testing: A Survey

Yoo and Harman [5] provide a comprehensive survey on regression testing techniques, covering three main areas: test case minimization, test case selection,

and test case prioritization. They evaluate both static and dynamic analysis techniques and provide classification of existing methods. One of the key contributions is the comparative analysis of strategies in terms of effectiveness, efficiency, and applicability in different contexts. This work does not propose a new method but rather shows the state-of-the-art up to that point, making it a strong reference for anyone researching regression testing optimization.

Zhao, Dua, and Singh (2019) – DeepTest: Testing Autonomous Systems

Zhao et al. [6] present *DeepTest*, a testing for deep neural networks (DNNs) used in autonomous driving systems. The authors recognize that traditional software testing methods fall short when applied to data-driven AI systems. DeepTest generates a made yet realistic driving scenarios by applying transformations (e.g., lighting changes, weather conditions) to input images. These transformations expose corner cases where the deep neural networks gives faulty results (e.g., incorrect steering decisions). This work contributes to the growing area of testing AI-based software and demonstrates how traditional coverage metrics can be extended to neural networks.

Qiu, Zhang, and Khurshid (2019) – Deep Reinforcement Learning for Software Testing

Qiu et al. [7] introduce deep reinforcement learning (DRL) for automated test case generation. Their key is modeling test generation as a reinforcement learning problem, where an agent interacts with the software under test and learns to explode all paths that are likely to expose bugs. The agent receives rewards for uncovering new program behaviors or causing failures, pushing it to intelligently prioritize areas of the code that are more error-prone. Their experimental results show that this method can outperform random testing and traditional methods in terms of coverage and fault detection. This work bridges the gap between AI planning techniques and software test automation.

3 Methodology

3.1 Data Collection and Feature Extraction

To demonstrate the effectiveness of machine learning models in test case selection, we built a dataset from a real-world open-source Java project, namely Apache Commons Collections, a widely-used library that offers reusable data structures for Java. The dataset was constructed by using the historical commits from the oldest SHA 15ad7824 to the newest SHA 6975b056 and extracting both static and dynamic characteristics of test classes across 122 revisions.

Repository and Version History. The Apache Commons Collections repository was cloned from its official Git source, extracting a list of commits was stored. Each commit represents a state of the system, Each commit was included only after verifying that the project built successfully without any errors, The commit selection was designed to ensure coverage across both minor and significant codebase changes, which is important for evaluating the impact on test selection.

Regression Test Selection with Ekstazi. We used Ekstazi, a well-established regression test selection tool for Java, which does a dependency analysis to reduce the set of tests to be rerun after code modifications. Ekstazi was programmatically integrated into the Maven build system by injecting it's profile plugin configuration into the pom.xml file. This integration enabled automatic test selection for each commit using the following steps:

1. The project was cleaned using `mvn clean` to ensure no residual artifacts that affects test selection.
2. Ekstazi was executed via `mvn ekstazi:ekstazi -Pekstazip`, which identifies the minimal set of tests impacted by the changes.
3. The selected test classes were pulled from `.ekstazi` directory within the same project.

This process was repeated for each consecutive pair of commits, simulating realistic regression testing scenarios in a continuous integration pipeline.

Feature Extraction. For each test class located under the standard Maven test directory (`src/test/java`), a set of features was extracted to be used later in the classification. These features aim to capture both static characteristics of the test class and contextual information based on changes in the codebase (Table 1).

Table 1. Extracted features and their descriptions

Feature	Description
`commit_hash`	Git commit identifier corresponding to the version of the project
`test_class`	Fully qualified name of the test class
`ekstazi_selected`	Binary target variable: 1 if the test was selected by Ekstazi, 0 otherwise
`test_class_length`	Total number of lines in the test class, used as a proxy for complexity
`num_test_methods`	Number of methods identified as test cases, either by `@Test` annotation or naming convention
`name_similar_to_changed`	Count of changed source classes whose names partially match the test class name
`unix_time`	Commit timestamp in Unix epoch format, indicating project evolution over time

The static analysis of test classes was performed using the `javalang` parser, which supports syntactic extraction of class and method-level constructs. Changes between code versions were identified using the `git diff` command, filtering only Java source files. The file paths were normalized into class-like representations to enable meaningful similarity comparisons between production and test code artifacts.

The resulting dataset was stored in a structured CSV file and served as the primary input for model training and evaluation in the upcoming stages.

3.2 Data Preprocessing

To prepare the dataset for model training and ensure its suitability for machine learning algorithms, we performed the following data preprocessing steps:

Encoding Categorical Features. Categorical features, such as *commit_hash* and *test_class*, were encoded into numerical representations using Label Encoding. This technique assigns a unique integer to each distinct category within a feature. Label Encoding was chosen for its simplicity and effectiveness in representing categorical data without introducing unnecessary dimensionality [8], as the order of categories is not relevant for our classification task. This transformation allows machine learning models to effectively process and interpret categorical information.

Scaling Numerical Features. Numerical features, including *test_class_length*, *num_test_methods*, *name_similar_to_changed*, and *unix_time*, were scaled using StandardScaler [9]. This technique standardizes numerical features by subtracting the mean and dividing by the standard deviation, resulting in features with zero mean and unit variance. Scaling is important to prevent features with larger ranges from dominating the model training process and ensures that all features contribute equally to the learning process, regardless of their original scales. This step enhances the performance and stability of machine learning models, particularly those sensitive to feature scaling, such as distance-based algorithms and gradient-based optimization methods.

Addressing Class Imbalance. The dataset exhibited a significant class imbalance, with a majority of test cases not selected by Ekstazi. To mitigate the potential bias introduced by this imbalance, we employed the Synthetic Minority Over-sampling Technique (SMOTE) [10]. SMOTE generates synthetic samples of the minority class (Ekstazi-selected test cases) by interpolating between existing minority class data points. This technique effectively balances the class distribution, preventing the model from being overly biased towards the majority class. We used SMOTE with its default parameters to create a balanced dataset, ensuring that both classes are adequately represented during model training and improving the model's ability to generalize to unseen data. These

preprocessing steps are essential to enhance the quality and suitability of the dataset for machine learning. Encoding categorical features allows models to handle categorical data effectively. Scaling numerical features ensures that all features contribute equally to the learning process and prevents features with larger ranges from dominating. Addressing class imbalance using SMOTE mitigates potential bias towards the majority class and improves the model's ability to generalize to unseen data. By carefully preprocessing the data, we aim to improve the performance, stability, and robustness of the machine learning models used for intelligent test case selection.

3.3 Feature Engineering

To potentially enhance the predictive power of the machine learning models, we performed feature engineering by creating new features from the existing ones. This process involved two main approaches:

Interaction Features. We engineered interaction features by multiplying pairs of existing features that were hypothesized to have a combined effect on Ekstazi's selection behavior. Specifically, we created the following interaction features:

- *interaction_1*: This feature represents the interaction between *test class length* and *num test methods*, capturing the potential influence of both the length of the test class and the number of test methods within it on test case selection.
- *interaction_2*: This feature represents the interaction between *name similar to changed* and *unix time*, aiming to capture the potential impact of the similarity between test names and changed code, combined with the temporal aspect of the commit.

These interaction features were included to allow models to learn more complex relationships between features that might not be captured by individual features alone.

Polynomial Features. To further explore non-linear relationships within the data, we utilized PolynomialFeatures to generate polynomial features of degree 2 from the numerical features (*test class length*, *num test methods*, *name similar to changed*, and *unix time*). This process creates new features by raising the original features to powers up to the specified degree, introducing quadratic terms and potentially capturing curvature in the relationships between features and the target variable.

Feature Selection. After generating the interaction and polynomial features, we performed feature selection to retain only the most relevant features for model training. This was done by calculating the absolute correlation between each feature and the target variable (Ekstazi selection) and removing features

with a correlation below a threshold of 0.05. This step aimed to reduce the dimensionality of the dataset, potentially improving model training efficiency and preventing overfitting by excluding features with minimal predictive power.

3.4 Model Selection and Training

For the task of test case selection, we investigated four widely used classification models: Logistic Regression [11], Random Forest [12], Support Vector Classifier (SVC) [13], and Gradient Boosting [14].

Model Selection. The selection of these models was based on their established performance in various classification tasks and their suitability for handling binary classification problems like test case selection. Each model offers a different approach to learning patterns from data, allowing us to explore a diverse range of algorithms and compare their effectiveness for our specific task.

Hyperparameter Tuning. To optimize the performance of each model, we performed hyperparameter tuning using GridSearchCV. This technique involves defining a grid of hyperparameter values for each model and systematically evaluating the model's performance using different combinations of these values. We used 5-fold cross-validation to ensure robust performance estimates during the tuning process. The F1-score was chosen as the evaluation metric for hyperparameter optimization, as it provides a good balance between precision and recall, which are crucial for test case selection.

Group-Based Splitting. To ensure realistic and unbiased performance estimates, we employed a group-based splitting strategy using GroupShuffleSplit. This technique ensures that data points belonging to the same commit (group) are either entirely in the training set or the testing set, preventing data leakage and maintaining the independence of test cases across different commits. This approach is a must for evaluating the models' ability to generalize to unseen commits, as it simulates the real-world scenario of applying the model to new code changes. The data was split into 80% for training and 20% for testing, and the training set was further divided into 75% for training and 25% for validation, maintaining the group-based splitting strategy throughout the process.

4 Results

This section presents the performance results of the four machine learning models evaluated for test case selection with machine learning: Logistic Regression, Random Forest, Support Vector Classifier (SVC), and Gradient Boosting. Each model was trained on the engineered and preprocessed dataset described in the previous sections, with evaluation conducted on a held-out test set using group-based splitting to ensure no data leakage across commits. Performance

was assessed using standard classification metrics: accuracy, precision, recall, F1-score, and Area Under the Receiver Operating Characteristic Curve (AUC-ROC). Additionally, Regression Test Selection (RTS) evaluation metrics, including safety violation, precision violation, and test suite reduction, were calculated to further assess the practical effectiveness of the models in reducing test suite sizes while maintaining safety.

4.1 Logistic Regression

The Logistic Regression model achieved an accuracy of 66.67%, with a high recall of 99.82% and a corresponding F1-score of 0.7980. However, the precision was relatively low at 66.47%, indicating a tendency to over-predict the positive class (Ekstazi-selected tests). The AUC-ROC score was 0.4621, suggesting limited ability to discriminate between classes across different thresholds. The classification report showed poor performance in identifying unselected test cases (class 0), with a recall of only 2%.

4.2 Random Forest

The Random Forest classifier attained an accuracy of 61.87%. Notably, it achieved a much higher precision of 83.06%, though this came at the cost of a lower recall of 53.01% compared to Logistic Regression. The F1-score was 0.6472, and the AUC-ROC improved significantly to 0.6688, indicating a better balance between sensitivity and specificity. Unlike Logistic Regression, the model showed greater ability to identify non-selected tests (class 0), achieving a recall of 79% in that category.

4.3 Support Vector Classifier (SVC)

The SVC model produced results similar to Logistic Regression, with an accuracy of 66.67%, precision of 66.43%, and a recall of 100%. This yielded the highest F1-score among all models at 0.7983. However, similar to Logistic Regression, SVC struggled to identify unselected tests, resulting in a recall of only 2% for class 0. AUC-ROC was not available due to the lack of probability scores in the default SVC configuration.

4.4 Gradient Boosting

Gradient Boosting achieved an accuracy of 62.34%, with a precision of 73.82% and a recall of 66.49%. Its F1-score was 0.6996, indicating solid performance across both classes. The AUC-ROC score was 0.6394, slightly lower than Random Forest but still indicative of reasonable classification capability. Gradient Boosting showed a more balanced treatment of both classes, with a 54% recall for class 0 and 66% for class 1.

4.5 Comparative Analysis

A summary of the results is shown in Table 2. While Logistic Regression and SVC achieved the highest recall and F1-scores, they suffered from poor ability to detect unselected tests, as reflected in their low AUC-ROC scores. Random Forest and Gradient Boosting, although slightly less accurate in terms of F1-score, offered more balanced performance and stronger discrimination ability, especially in identifying both positive and negative test cases. These findings highlight the trade-off between aggressive positive class prediction and balanced generalization.

Table 3 presents the RTS evaluation metrics for each model. Random Forest achieved the highest test suite reduction with relatively low safety violation [15], making it the best choice in terms of minimizing test suite size while maintaining a high level of safety. SVC and Logistic Regression had lower test suite reductions and higher safety violations. Gradient Boosting, while offering balanced performance, did not outperform Random Forest in terms of test suite reduction and experienced higher precision violations [15].

Table 2. Classification performance comparison of models

Model	Accuracy	Precision	Recall	F1-score	AUC-ROC
Logistic Regression	0.667	0.665	**0.998**	0.798	0.462
Random Forest	0.619	**0.831**	0.530	0.647	**0.669**
Support Vector Classifier	0.667	0.664	**1.000**	**0.798**	N/A
Gradient Boosting	0.623	0.738	0.665	0.700	0.639

Table 3. RTS evaluation metrics comparison of models

Model	Safety Violation	Precision Violation	Test Suite Reduction
Logistic Regression	0.3349	0.0012	0.0094
Random Forest	0.0976	0.4240	**0.5789**
Support Vector Classifier	0.3357	0.0000	0.0070
Gradient Boosting	0.1908	0.2712	0.4058

5 Conclusion

The evaluation of four machine learning models for intelligent test case selection—Logistic Regression, Random Forest, Support Vector Classifier (SVC), and Gradient Boosting—reveals distinct trade-offs in terms of performance, safety, and precision violations, as well as test suite reduction.

Logistic Regression and SVC both exhibited high safety violations, indicating that they are selecting a significant number of tests that Ekstazi did not deem

necessary. While both models showed very low precision violations, their test suite reductions were minimal, suggesting that they are rather conservative in their approach. This behavior might be beneficial in situations where minimizing the risk of missing critical tests is a priority, but it results in lower efficiency in terms of reducing the test suite size.

On the other hand, Random Forest achieved the highest test suite reduction but at the cost of a high precision violation. This implies that Random Forest aggressively reduces the test suite, potentially missing important tests that Ekstazi considers essential. This trade-off could be risky, particularly in scenarios where missing critical functionality could lead to significant issues.

Gradient Boosting struck a balance between the extremes. It provided a reasonable test suite reduction while maintaining moderate safety and precision violations. This suggests that Gradient Boosting is neither too aggressive nor too conservative, making it a well-rounded choice for test case selection.

In conclusion, Gradient Boosting appears to be the best overall choice for intelligent test case selection in this specific scenario. It offers a balanced approach that reduces the test suite without significantly compromising the accuracy of test case selection. However, the selection of the optimal model depends on several factors, including:

Risk Tolerance: If the project has a very low tolerance for missing critical bugs, models like Logistic Regression or SVC, with their low precision violations, might be more suitable, despite their limited test suite reduction.

Computational Resources: If reducing execution time is a priority and some risk of missing tests is acceptable, Random Forest could be a viable choice due to its high test suite reduction.

Context: Ultimately, the "best" model is context-dependent. The trade-offs between safety, precision, and test suite reduction should be carefully weighed against the specific needs and priorities of the project. Therefore, it's essential to consider the risk appetite and project objectives before making a final decision.

References

1. Harrold, M.J., et al.: Regression test selection for Java software. In: OOPSLA, pp. 312–326 (2001)
2. Orso, A., Shi, N., Harrold, M.J.: Scaling regression testing to large software systems. In: FSE, pp. 241–251 (2004)
3. Gligoric, M., Eloussi, L., Marinov, D.: Practical regression test selection with dynamic file dependencies. In: ISSTA, pp. 211–222 (2015)
4. Larsson, J., and Höstklint, N.: Dynamic test case selection using machine learning. Master's thesis, Linnaeus University (2021)
5. Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: a survey. Softw. Test. Verificat. Reliabil. **22**(2), 67–120 (2012)
6. Zhao, Y., Dua, D., and Singh, S.: DeepTest: automated testing of deep-neural-network-driven autonomous cars. In: Proceedings of the 40th International Conference on Software Engineering (2019)

7. Qiu, X., Zhang, L., Khurshid, S.: A deep reinforcement learning framework for software testing. In: Proceedings of the 41st International Conference on Software Engineering (ICSE), pp. 347–358 (2019)
8. Pedregosa, F., et al.: Scikit-learn: machine learning in python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)
9. Scikit-learn: Preprocessing data—StandardScaler. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html
10. Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P.: SMOTE: synthetic minority over-sampling technique. J. Artif. Intell. Res. **16**, 321–357 (2002)
11. Scikit-learn: Logistic Regression. https://scikit-learn.org/stable/modules/generated/sklearn.linear$_i$model.LogisticRegression.html
12. Scikit-learn: Random Forest Classifier. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html
13. Scikit-learn: Support Vector Classification. https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html
14. Scikit-learn: Gradient Boosting Classifier. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html
15. Sirin, E., and Gotlieb, A.: Ekstazi: cost-effective and accurate regression testing in the large. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, pp. 347–357 (2016)