Journal of Applied Informatics and Computing (JAIC)
Vol.10, No.1, February 2026, pp. 224~231
e-ISSN: 2548-6861

SemetonBug: Next-Generation Machine Learning-Powered Code Analyzer for Precision Bug Detection and Dynamic Error Localization

Surni Erniwati [1*], Bahtiar Imran [2], Zumratul Muahidin [3***], Zaeniah [4***], Juhartini [5****]**
[*] Manajemen Informatika, Universitas Teknologi Mataram
[**] Rekayasa Sistem Komputer, Universitas Teknologi Mataram
[***] Sistem Informasi, Universitas Teknologi Mataram
[****] Teknik Informatika, Universitas Teknologi Mataram
mentari1990@gmail.com [1], bahtiarimranlombok@gmail.com [2], muahidinzumratul@gmail.com [3], zaen1989@gmail.com [4], juhartini8815@gmail.com [5]

Article Info

Article history:

Received 2025-11-24
Revised 2025-12-22
Accepted 2026-01-07

Keyword:

Bug Detection,
Machine Learning,
Python,
Random Forest,
Abstract Syntax Tree.

ABSTRACT

Bug detection in Python programming is a crucial challenge in software development. This research proposes SemetonBug, a machine learning-based system for automatically detecting bugs in Python code. The system utilizes a Random Forest Classifier as the main model, with features extracted from the syntactic structure of the code using an Abstract Syntax Tree (AST). The dataset consists of 200 Python files, divided into 100 files with bugs and 100 files without bugs. The model is optimized using Grid Search Cross Validation, with the best combination of n_estimators = 300, max_depth = 20, min_samples_split = 5, and min_samples_leaf = 2. Evaluation results show that the model achieves 85% accuracy, 0.84 precision, 0.87 recall, and 0.86 F1-score. The detected bugs are stored in an Excel file for further analysis. By leveraging machine learning, SemetonBug enhances efficiency and accuracy in bug identification compared to traditional rule-based methods. These findings highlight the potential of machine learning models in improving software quality and reducing coding errors automatically.

I. INTRODUCTION

Bug detection in Python programming is one of the most critical challenges in software development. Various studies have demonstrated that bugs can be minimized through appropriate tools and techniques. For instance, PYBUGLAB, an implementation designed for the Python language, is developed to detect and fix various types of simple bugs that significantly impact code accuracy. By focusing on simple bugs, this tool illustrates that minor corrections can lead to substantial improvements in overall code quality, reducing errors and enhancing software reliability [1]. Empirical analysis of code modifications after bug fixes in Python reveals specific patterns, indicating that changes are not made randomly but occur within a particular coding context [2]. This finding underscores the importance of understanding code context for further advancements in bug detection tools. In the modern era, advanced techniques are increasingly employed, including machine learning models for bug detection and duplicate bug report identification. Research indicates that attention-based models can improve accuracy in detecting duplicate bug reports, which often pose challenges in large-scale software management [3]. In software development, manual bug identification is often time-consuming, prone to human error, and inefficient, particularly when dealing with large-scale codebases. This manual approach relies on programmers' expertise to review and test code directly, which may result in undetected bugs or issues only discovered during the final testing phase. Therefore, automated methods that enable faster, more accurate, and consistent bug detection are essential.

Several previous studies have been conducted to detect bugs in program code, employing both static approaches such as code analysis and machine learning-based methods to enhance detection accuracy. Among them, [4] explores machine learning techniques for software bug prediction, demonstrating that ensemble algorithms outperform individual approaches. Study [5] investigates the application

of AI in software development practices, including bug detection, which improves efficiency and software quality. Meanwhile, [6] focuses on understanding bugs within a multilingual deep learning framework, forming a crucial foundation for detecting and resolving bugs in AI-based applications. Study [7] introduces APIScanner, a tool that automates the detection of deprecated APIs in Python libraries, contributing to improved code quality. Research [8] discusses machine learning techniques for software bug prediction, showcasing the development of several effective prediction models. In [9] artificial immune networks are applied to optimize hyperparameters in bug prediction classification models, aiming to enhance the software testing process. Study [10] compares cellular automata implementations using image processing and machine learning for code validation, albeit with a broader focus on general verification methods. Meanwhile, [1] develops BUGLAB, a self-supervised approach for bug detection and repair, highlighting the potential of machine learning-based methods in software development. Study [11] examines the use of code embeddings and transformers to assist programming tasks, including bug detection, underscoring AI's critical role in software development. Meanwhile, [12] describes a Python package called sstar, though its focus is more on genetic analysis rather than software bug detection. In [13], an automated bug report detection and classification system using deep learning techniques is introduced, demonstrating improvements in software management speed. Study [14] investigates entropy-based machine learning models for assessing bug severity, emphasizing the importance of bug descriptions in predicting other attributes. Meanwhile, [15] utilizes large datasets to train models for filtering code warnings, relevant to static code analysis and bug detection. Research [16] conducts an empirical study on bugs in COVID-19 software projects, identifying the need for better detection tools for security monitoring. Study [17] proposes BPDET, a bug prediction model leveraging deep representation techniques and ensemble learning, offering an in-depth analysis of bug detection improvements. Study [18] examines user interface design in AI-based Python visual applications, though its focus lies in UI design rather than bug detection. Lastly, [19] evaluates smart contract analysis tools for bug detection, serving as an example of static techniques for identifying software errors.

This study aims to develop a bug detection system for Python code, named SemetonBug. The system is designed to enhance efficiency in automatically identifying code errors using machine learning methods, specifically Random Forest as the primary classification model. Unlike traditional rule-based static analysis approaches, SemetonBug employs machine learning based on abstract features extracted from the syntactic structure of the code to detect various types of bugs. The source code used in this study was manually collected, comprising 200 Python files, evenly divided into 100 files containing bugs and 100 bug-free files. Each file was analyzed using the Abstract Syntax Tree (AST) library to extract relevant features such as the number of statements within loops, the depth of if-else structures, variable usage, and function call patterns. SemetonBug is designed to detect bugs simultaneously, enabling the identification of multiple errors in a single analysis without requiring manual inspection of each instance. In training the model, Random Forest was chosen for its ability to handle complex features and provide stable classification results. The model was optimized using hyperparameter tuning via Grid Search, with explored parameters including n_estimators: [10, 50, 100, 200], max_depth: [None, 10, 20, 30], min_samples_split: [2, 5, 10], and min_samples_leaf: [1, 2, 4]. Model evaluation was conducted using multiple metrics, including accuracy, precision, recall, and F1-score. Additionally, performance analysis was carried out using a Confusion Matrix, ROC Curve, and Learning Curve to assess how the model learns as the training data increases. The bug detection results from SemetonBug are stored in Excel format, allowing software developers to perform further analysis on the identified errors. Through this approach, the study introduces an innovative machine learning-based bug detection method, replacing traditional rule-based static analysis techniques commonly used in conventional bug detection systems.

II. METHOD

A. Dataset Preparation

The dataset in this study consists of 200 Python code files, comprising 100 bugged files (containing bugs) and 100 non-bugged files (bug-free). The dataset was manually collected from various sources, including self-written code, open-source projects, and programming forums. Bugged code contains syntax errors (e.g., unbalanced parentheses), logical errors (algorithmic mistakes), or runtime errors (such as division by zero), whereas non-bugged code executes without errors. Before use, the dataset undergoes preprocessing, which includes three main stages. First, Python code tokenization is performed to convert the code into tokens according to Python syntax. Second, comments and excessive whitespace are removed to ensure that the model analyzes only the relevant code. Third, variable and function name normalization replaces variable and function names with generic tokens (e.g., var1, func1, etc.) to prevent bias caused by naming patterns. With this preprocessing, the dataset becomes cleaner and more structured for feature extraction, where various Python code characteristics are extracted as input for the machine learning model to detect bugs automatically.

B. Feature Extraction

In this study, feature extraction is performed to obtain a numerical representation of Python code characteristics that may indicate the presence of bugs. These features are selected based on structural aspects of the code that potentially influence complexity and the likelihood of errors. One of the primary features used is the number of functions within the code, as a higher number of defined functions increases the structural complexity, thereby elevating the risk of bugs.

Additionally, the number of classes is also considered, particularly in Object-Oriented Programming (OOP)-based code, where interclass relationships can complicate code comprehension and maintenance. Moreover, the number of declared variables serves as a crucial feature, as inconsistent variable usage may lead to execution errors. The total lines of code are also taken into account as an indicator of complexity, with longer code generally having more bug-prone points. Another important feature is the count of conditional statements, such as if, elif, and else, which are frequently used to control program execution flow. A higher number of conditions increases the likelihood of logical errors due to improper condition handling. Additionally, this study considers the number of exception-handling statements, determined by counting occurrences of try-except blocks. Excessive or insufficient exception handling may indicate potential bugs within the code. All these features are automatically extracted using Abstract Syntax Tree (AST) [20], which enables the analysis of code structure without requiring program execution. The data obtained from these features is then used as input for the machine learning model to distinguish between bugged and non-bugged code.

C. Machine Learning Model

In this study, the Random Forest Classifier algorithm is employed to develop a bug detection model for Python code. Random Forest is chosen due to its ability to handle datasets with complex features and its robustness against overfitting. This algorithm constructs multiple decision trees and aggregates their results to enhance prediction accuracy [20]–[23]. The preprocessed dataset is split into two subsets: a training set (70%) and a testing set (30%), using stratified splitting to maintain class balance. The model is trained using Grid Search with cross-validation (cv=5) to optimize hyperparameters such as the number of trees in the forest (n_estimators), maximum tree depth (max_depth), minimum samples required for node splitting (min_samples_split), minimum samples per leaf (min_samples_leaf), and the best feature selection method at each split (max_features). After training, the model's performance is evaluated using accuracy, precision, recall, and F1-score metrics to assess the balance between true positive and negative predictions. Additionally, a confusion matrix is utilized to analyze the distribution of model errors. With this approach, the resulting model is expected to detect bugs in Python code with optimal accuracy.

The primary function in Random Forest:
Prediction based on majority voting from decision trees.

$$y = mode\{h_1(x), h_2(x), \ldots, h_T(x)\} \quad (1)$$

with $\boldsymbol{h_i}(\boldsymbol{x})$ The prediction result from the -i tree, and $\boldsymbol{T}$ is the total number of trees in the model.

Node splitting function in Decision Tree using Gini Impurity:

$$Gini = 1 - \sum_{i=1}^{n} p2i \quad (2)$$

where p_i is the probability of a class within the node.

D. Hyperparameter Optimization

To achieve optimal performance of the Random Forest Classifier model, hyperparameter optimization is conducted using the Grid Search method with 5-fold cross-validation (cv=5). Grid Search systematically evaluates various combinations of hyperparameter values to identify the best configuration that yields the highest accuracy. The adjusted hyperparameters include:

1. n_estimators (number of trees in the forest): [100, 300, 500, 1000]
2. max_depth (maximum depth of decision trees): [None, 10, 20, 30, 50, 100]
3. min_samples_split (minimum number of samples required to split a node): [2, 5, 10, 20]
4. min_samples_leaf (minimum number of samples required at each leaf node): [1, 2, 4, 10]
5. max_features (number of features considered for each split): ['sqrt', 'log2', None]

During the optimization process, Grid Search systematically evaluates all combinations of the above parameters and selects the configuration that provides the best performance based on the average accuracy obtained from cross-validation. By employing 5-fold cross-validation, the dataset is divided into five parts, where the model is trained on four parts and tested on the remaining part in an iterative manner until the entire dataset is utilized for validation.

The formula used is[24], [25]:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Samples}} = \frac{TP + TN}{TP + TN + FP + FN} \quad (3)$$

Where:

- TP = True Positives (correct predictions for the bugged class).
- TN = True Negatives (correct predictions for the non-bugged class).
- FP = False Positives (incorrect predictions for the bugged class).

FN = False Negatives (incorrect predictions for the non-bugged class).

III. RESULTS AND DISCUSSION

A. Best Model Selection

In the process of selecting the best model, an exploration of various hyperparameter combinations was conducted using the Grid Search Cross Validation (GridSearchCV) method on the Random Forest Classifier model. The parameter grid tested includes variations in the number of decision trees (n_estimators), the maximum depth of the trees (max_depth), the minimum number of samples required to split a node

(min_samples_split), and the minimum number of samples required at a leaf node (min_samples_leaf). After performing five-fold cross-validation, the best model was obtained with an accuracy of 85% on the test data. This high-performing model utilizes the hyperparameter combination n_estimators = 300, max_depth = 20, min_samples_split = 5, and min_samples_leaf = 2. The model selection was based on the mean_test_score, where this specific hyperparameter configuration yielded the highest accuracy compared to other combinations. With 300 decision trees, the model effectively captures complex patterns in the data without overfitting. Restricting the maximum depth of the trees to 20 levels helps control model complexity, while min_samples_split = 5 and min_samples_leaf = 2 ensure that node splitting and the number of samples in leaf nodes remain optimal for improving model generalization on new data. Beyond accuracy, the model was also evaluated using Precision, Recall, and F1-score to provide a more comprehensive assessment of its classification performance. The selected model achieved a Precision of 0.84, Recall of 0.87, and F1-score of 0.86. Precision indicates that 84% of all positive predictions made by the model were correct, meaning the model has a relatively low false positive rate. High precision is particularly crucial in scenarios where incorrect positive classifications have serious consequences, such as fraud detection or disease diagnosis. Meanwhile, Recall of 87% suggests that 87% of actual positive instances were correctly identified by the model. A high recall is essential in situations where missing positive cases could be critical, such as in security systems or intrusion detection. The F1-score of 0.86, which represents the harmonic mean of precision and recall, demonstrates that the model achieves a well-balanced trade-off between detecting positive cases and ensuring prediction accuracy. With an accuracy of 85%, along with high precision, recall, and F1-score, the model exhibits strong classification performance with relatively low error rates. This indicates that the model is highly reliable in identifying and classifying data correctly, making it a suitable approach for tasks requiring accurate and robust classification capabilities.

Figure 1 presents the Confusion Matrix of the model's classification performance in distinguishing between two categories: Non-Bugged and Bugged. This matrix provides insights into the number of correct and incorrect predictions made by the model. Within the matrix, 24 samples were correctly classified as Non-Bugged (True Negative), while 5 Non-Bugged samples were misclassified as Bugged (False Positive). For the Bugged category, the model successfully identified 27 samples correctly (True Positive), but 4 Bugged samples were incorrectly classified as Non-Bugged (False Negative). Based on this confusion matrix, key evaluation metrics were computed. Precision, which quantifies the model's accuracy in identifying the Bugged class, is calculated as 27 / (27 + 5) = 0.84 (84%), indicating that 84% of the positive predictions were correct. Recall, reflecting the model's ability to capture all truly Bugged samples, is 27 / (27 + 4) = 0.87 (87%), signifying that 87% of the total Bugged samples were correctly identified.

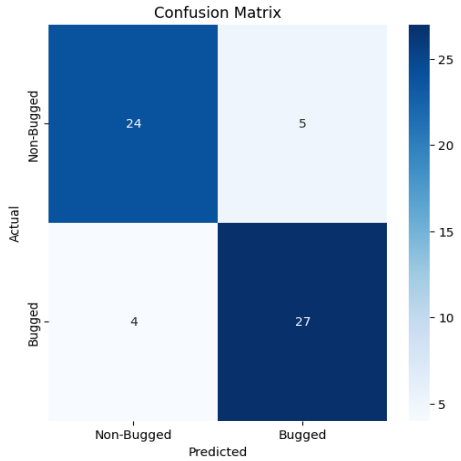

Figure. 1. Confusion Matrix of Testing Results

The F1-score, which represents the harmonic mean of precision and recall, is 0.86, demonstrating a well-balanced trade-off between model precision and sensitivity. Overall, the model exhibits strong classification performance with a relatively low error rate, indicating its reliability in distinguishing between Bugged and Non-Bugged samples.

B. Bug Detection Results

The evaluation results of the bug detection system demonstrate a satisfactory performance in identifying errors within the code. With an accuracy rate of 85%, the system effectively detects bugs with a precision of 0.84, recall of 0.87, and an F1-score of 0.86. The higher recall compared to precision indicates that the system is proficient in identifying most of the existing bugs, although some false positives are still present. Overall, these results suggest that the applied method is reasonably reliable in detecting and classifying bugs. However, there remains room for improvement, particularly in reducing misclassification errors and enhancing the system's efficiency. A representative example of the bug detection results is illustrated in Figure 2.

Figure 2 illustrates an example of the bug detection results performed by SemetonBug in classifying Python code files containing errors. Each output line generated by the system provides information regarding the analyzed file, the model's prediction, and the actual condition of the file. Each entry follows a structured format, including the file name, the system's prediction, the actual condition of the file, and the specific lines of code that contain bugs. For instance, in the case of filebug (87).py, the system predicts that the file contains a bug (Bug Detected), which aligns with its actual condition (bugged). The specific lines identified as containing errors in this file are lines 11, 12, 13, 15, and 17.

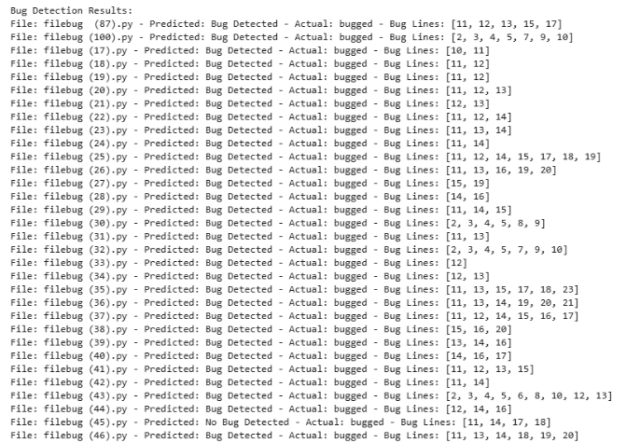

Figure. 2. Example of Python Bug Detection

The detection results indicate that the system is capable of identifying bugs across multiple files simultaneously, while also providing the corresponding line numbers where the errors occur. This feature facilitates a more systematic review and debugging process for developers. Additionally, an edge case is observed in filebug (45).py, where the system predicts the absence of bugs (No Bug Detected), while in reality, the file still contains errors. This discrepancy suggests the presence of false negatives, indicating that certain bugs might remain undetected by the model. The complete detection results can be further examined in Table 1

TABLE 1.
BUG DETECTION TESTING RESULTS

No	Filename	Predicted Label	Actual Label	Bug Lines
1	filebug (87).py	Bug Detected	bugged	11, 12, 13, 15, 17
2	filebug (100).py	Bug Detected	bugged	2, 3, 4, 5, 7, 9, 10
3	filebug (17).py	Bug Detected	bugged	10, 11
4	filebug (18).py	Bug Detected	bugged	11, 12
5	filebug (19).py	Bug Detected	bugged	11, 12
6	filebug (20).py	Bug Detected	bugged	11, 12, 13
7	filebug (21).py	Bug Detected	bugged	12, 13
8	filebug (22).py	Bug Detected	bugged	11, 12, 14
9	filebug (23).py	Bug Detected	bugged	11, 13, 14
10	filebug (24).py	Bug Detected	bugged	11, 14
11	filebug (25).py	Bug Detected	bugged	11, 12, 14, 15, 17, 18, 19
12	filebug (26).py	Bug Detected	bugged	11, 13, 16, 19, 20
…	…	…	…	…
…	…	…	…	…
200	filenobug(4).py	No Bug Detected	non_bugged	None

The bug detection testing results are presented in Table 1, illustrating the system's performance in identifying errors across various code files. The table consists of four primary columns: Filename, Predicted Label, Actual Label, and Bug Lines. The Filename column lists the names of the tested files. The Predicted Label column indicates the system's classification, where "Bug Detected" is assigned if the system identifies an error and "No Bug Detected" if no issues are found. The Actual Label column provides the ground truth for each file, where "bugged" indicates the presence of a bug, while "non-bugged" confirms the absence of errors. The Bug Lines column specifies the exact lines of code where the detected errors occur. If a file is classified as bugged, the system provides the corresponding line numbers where issues are found. Conversely, for non-bugged files, the system returns "No Bug Detected" without listing any lines. The results indicate that the system effectively identifies bugs across multiple files, accurately pinpointing the specific lines where errors occur. Furthermore, in cases where no bugs are present, the system successfully classifies the files, demonstrating its ability to distinguish between buggy and non-buggy files. However, further analysis is required to assess potential misclassifications, particularly false positives and false negatives, which may impact the system's overall effectiveness.

C. ROC Curve Analysis

The Receiver Operating Characteristic (ROC) Curve is an evaluation tool used to analyze the performance of the bug detection system by considering the trade-off between the True Positive Rate (TPR) and the False Positive Rate (FPR). The ROC Curve provides insights into the model's ability to distinguish between files that contain bugs (bugged) and those that do not (non-bugged).

Figure 3 illustrates the Receiver Operating Characteristic (ROC) Curve, which is used to evaluate the performance of the bug detection system. The ROC Curve visualizes the relationship between the True Positive Rate (TPR) and the False Positive Rate (FPR) across different decision thresholds. The blue curve in the graph represents the performance of the bug detection model, while the gray diagonal line serves as a reference baseline for a random model with AUC = 0.5.

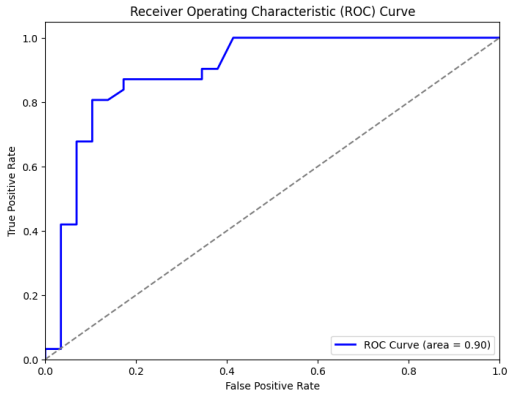

Figure. 3. Receiver Operating Characteristic (ROC)

Based on the evaluation results, the model achieves an Area Under the Curve (AUC) of 0.90, indicating a highly effective classification capability in distinguishing between bugged and non-bugged files. A higher AUC value suggests superior model performance, with strong discriminative power. This AUC score demonstrates that the system maintains a well-balanced trade-off between sensitivity and specificity, making it a reliable tool for bug detection in program code.

D. Learning Curve and Model Generalization

A learning curve is a graphical representation that illustrates how a model's performance evolves as the amount of training data increases. This curve is used to assess whether the model suffers from overfitting, underfitting, or has a good generalization capability. By analyzing the learning curve, it is possible to determine whether the model benefits from additional training data or if adjustments to hyperparameters are necessary to enhance its performance.

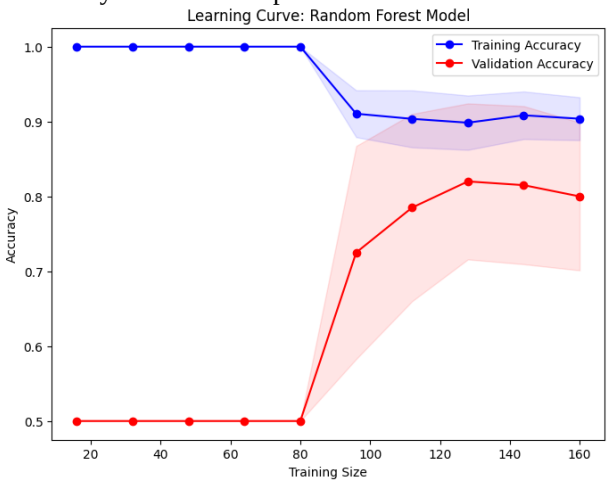

Figure. 4. Learning Curve of Random Forest

Figure 4 illustrates the learning curve of the Random Forest model, depicting the relationship between the training dataset size and the model's accuracy on both training and validation data. The blue curve represents training accuracy, while the red curve represents validation accuracy. In the initial phase, when the training dataset is small, the model exhibits high training accuracy (close to 100%), whereas the validation accuracy remains relatively low (around 50%). This phenomenon indicates overfitting, where the model overly adapts to the training data but struggles to generalize to unseen data. As the training dataset grows, validation accuracy starts to improve, while training accuracy slightly decreases, reaching a balance of approximately 90% for training and 80% for validation. This suggests that the model is achieving better generalization, where its performance on unseen data becomes more stable. However, a large variance in the validation curve is observed, particularly during the increasing phase of the training data. This variability may be attributed to model complexity or data imbalance. Further adjustments, such as regularization techniques or increasing the training dataset size, could help mitigate this fluctuation and enhance the model's stability in bug detection.

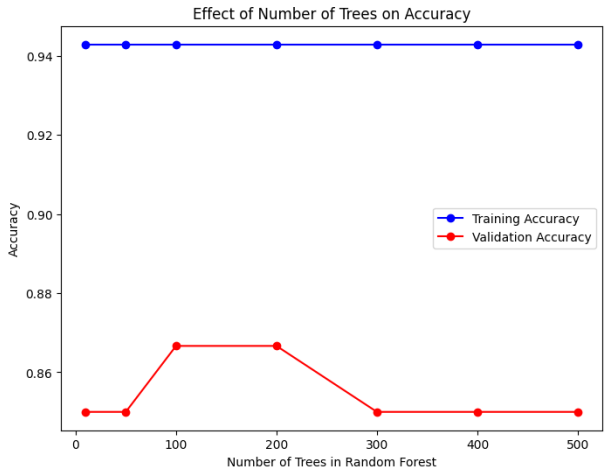

Figure. 5. Analysis of the Effect of the Number of Trees in the Random Forest Algorithm on Model Accuracy

Figure 5 illustrates the impact of the number of trees in the Random Forest model on accuracy for both training and validation data. The blue curve represents training accuracy, while the red curve represents validation accuracy. It can be observed that the training accuracy remains consistently high and stable at approximately 94%, with no significant changes despite the increase in the number of trees. This indicates that the model learns effectively from the training data without experiencing performance degradation. However, fluctuations are observed in the validation accuracy. When the number of trees is within the range of 100 to 200, validation accuracy reaches its peak value of approximately 86%. Beyond this point, a gradual decline is noticeable. For tree counts exceeding 300, validation accuracy tends to stabilize around 85%, suggesting that increasing the number of trees does not necessarily enhance the model's generalization ability. This decline in validation accuracy may be attributed to overfitting, where the model becomes overly complex, adapting too closely to the training data while failing to capture meaningful patterns for unseen data.

E. Discussion

The Receiver Operating Characteristic (ROC) curve generated illustrates the model's ability to distinguish between positive and negative classes based on predicted probabilities. From the displayed graph, it is evident that the model's ROC curve is positioned well above the diagonal line (baseline random classifier), indicating superior classification performance compared to random guessing. The Area Under the Curve (AUC) value of 0.90 suggests that the model has a high accuracy in differentiating between classes, although there is still room for improvement. The closer the AUC value is to 1, the better the model performs in classification with minimal false positives.

The learning curve provides further insight into the model's generalization capacity on new data. From the graph, the training accuracy (blue line) is initially high, while the validation accuracy (red line) gradually increases as the training dataset expands. Initially, a significant gap exists between the training and validation accuracy, indicating potential overfitting when the model is trained with limited data. However, as the training data increases, this gap begins to narrow, showing that the model becomes better at learning patterns without overly relying on training data. In other words, the model's generalization ability improves, although fluctuations in validation accuracy suggest further optimization potential, such as hyperparameter tuning or regularization techniques.

Furthermore, the analysis of the number of trees in the Random Forest algorithm reveals that while training accuracy remains consistently high, validation accuracy exhibits slight fluctuations as the number of trees increases. This indicates that increasing the number of trees in a Random Forest model does not always significantly enhance its performance, particularly beyond a certain threshold. The graph suggests that after reaching a certain number of trees, validation accuracy stagnates or even declines, which may be attributed to the diminishing returns effect. Beyond this point, additional trees provide little performance improvement while increasing computational complexity without a proportional accuracy gain. Thus, determining the optimal number of trees is crucial for balancing accuracy and computational efficiency.

Overall, the results indicate that the model demonstrates strong classification performance, yet several aspects can still be improved. Further optimization efforts should focus on minimizing overfitting, balancing bias and variance, and fine-tuning hyperparameters to achieve better results. Additionally, techniques such as ensemble learning, feature selection, or increasing the training dataset size can further enhance the model's performance, particularly in more complex scenarios.

IV. CONCLUSION

In this study, the Random Forest Classifier model was explored for bug detection, utilizing Grid Search Cross-Validation to select the optimal hyperparameters. The selected model achieved an accuracy of 85% with the following hyperparameter configuration: n_estimators = 300, max_depth = 20, min_samples_split = 5, and min_samples_leaf = 2. Performance evaluation indicated that the model achieved a precision of 0.84, recall of 0.87, and an F1-score of 0.86, demonstrating a well-balanced trade-off between precision and sensitivity in bug classification.

The confusion matrix analysis revealed that the model was able to identify bugs effectively, although some misclassifications (false positives and false negatives) were still present. The ROC Curve with an AUC of 0.90 further indicated that the model exhibited strong discrimination capability between buggy and non-buggy files. Additionally, the learning curve analysis suggested that the model generalized well as the training data increased, despite a slight indication of overfitting when the number of trees became excessively large.

Overall, the proposed approach proved to be effective in detecting bugs in source code with a high performance level. However, there is still room for improvement, such as further hyperparameter optimization, exploring other ensemble methods, or applying regularization techniques to mitigate overfitting. Future research could also consider the integration of deep learning techniques to enhance bug detection accuracy while reducing classification errors.

REFERENCES

[1] M. Allamanis, H. Jackson-Flux, and M. Brockschmidt, "Self-Supervised Bug Detection and Repair," in *Journal of Mathematical Sciences*, 2021. doi: 10.48550/arxiv.2105.12787.

[2] D. Cotroneo, L. De Simone, A. K. Iannillo, R. Natella, S. Rosiello, and N. Bidokhti, "Analyzing the Context of Bug-Fixing Changes in the OpenStack Cloud Computing Platform," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, 2019. doi: 10.1109/issre.2019.00041.

[3] M. Ben Messaoud, A. Miladi, I. Jenhani, M. W. Mkaouer, and L. Ghadhab, "Duplicate Bug Report Detection Using an Attention-Based Neural Language Model," *Ieee Trans. Reliab.*, 2023, doi: 10.1109/tr.2022.3193645.

[4] S. N. Saharudin, T. W. Koh, and S. N. Kew, "Machine Learning Techniques for Software Bug Prediction: A Systematic Review," *J. Comput. Sci.*, 2020, doi: 10.3844/jcssp.2020.1558.1569.

[5] D. Ajiga, P. A. Okeleke, S. O. Folorunsho, and C. Ezeigweneme, "Enhancing Software Development Practices With AI Insights in High-Tech Companies," *Comput. Sci. \& It Res. J.*, 2024, doi: 10.51594/csitrj.v5i8.1450.

[6] Z. Li, S. Wang, W. Wang, P. Liang, R. Mo, and B. Li, "Understanding Bugs in Multi-Language Deep Learning Frameworks," *Ieee Access*, 2023, doi: 10.48550/arxiv.2303.02695.

[7] A. Vadlamani, R. Kalicheti, and S. Chimalakonda, "APIScanner -- Towards Automated Detection of Deprecated APIs in Python Libraries," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2021. doi: 10.48550/arxiv.2102.09251.

[8] N. A. Adam Khleel and K. Nehéz, "Comprehensive Study on Machine Learning Techniques for Software Bug Prediction," *Int. J. Adv. Comput. Sci. Appl.*, 2021, doi:

10.14569/ijacsa.2021.0120884.

[9] F. Khan, S. Kanwal, S. Alamri, and B. Mumtaz, "Hyper-Parameter Optimization of Classifiers, Using an Artificial Immune Network and Its Application to Software Bug Prediction," *Ieee Access*, 2020, doi: 10.1109/access.2020.2968362.

[10] M. K. Wozniak and P. J. Giabbanelli, "Comparing Implementations of Cellular Automata as Images: A Novel Approach to Verification by Combining Image Processing and Machine Learning," in *SIGSIM-PADS '21*, 2021. doi: 10.1145/3437959.3459256.

[11] S. Kotsiantis, V. S. Verykios, and M. Tzagarakis, "AI-Assisted Programming Tasks Using Code Embeddings and Transformers," *Electronics*, 2024, doi: 10.3390/electronics13040767.

[12] X. Huang, P. Kruisz, and M. Kuhlwilm, "Sstar: A Python Package for Detecting Archaic Introgression From Population Genetic Data With S*," *Mol. Biol. Evol.*, 2022, doi: 10.1101/2022.03.10.483765.

[13] A. Kukkar, R. Mohana, Y. Kumar, A. Nayyar, M. Bilal, and K. S. Kwak, "Duplicate Bug Report Detection and Classification System Based on Deep Learning Technique," *IEEE Access*, vol. 8, pp. 200749–200763, 2020, doi: 10.1109/ACCESS.2020.3033045.

[14] M. Kumari, U. K. Singh, and M. Sharma, "Entropy Based Machine Learning Models for Software Bug Severity Assessment in Cross Project Context," *Comput. Sci. Its Appl.*, 2020, doi: 10.1007/978-3-030-58817-5_66.

[15] P. Hegedűs and R. Ferenć, "Static Code Analysis Alarms Filtering Reloaded: A New Real-World Dataset and Its ML-Based Utilization," *Ieee Access*, 2022, doi: 10.1109/access.2022.3176865.

[16] A. Rahman and E. Farhana, "An Empirical Study of Bugs in COVID-19 Software Projects," *J. Softw. Eng. Res. Dev.*, 2021, doi: 10.5753/jserd.2021.827.

[17] S. K. Pandey, R. B. Mishra, and A. K. Tripathi, "BPDET: An Effective Software Bug Prediction Model Using Deep Representation and Ensemble Learning Techniques," *Expert Syst. Appl.*, 2020, doi: 10.1016/j.eswa.2019.113085.

[18] U. Dikme, "Industrial User Interface Software Design for Visual Python AI Applications Using Embedded Linux Based Systems," *J. Appl. Phys. Sci.*, 2021, doi: 10.20474/japs-7.1.

[19] A. Ghaleb and K. Pattabiraman, "How Effective Are Smart Contract Analysis Tools? Evaluating Smart Contract Static Analysis Tools Using Bug Injection," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020. doi: 10.1145/3395363.3397385.

[20] K. Bharath and P. Jagadeesh, "An Innovative Software Bug Prediction System using Random Forest Algorithm for Enhanced Accuracy in Comparison with Logistic Regression Algorithm," in *2023 Intelligent Computing and Control for Engineering and Business Systems (ICCEBS)*, 2023.

[21] S. T. Cynthia, B. Roy, and D. Mondal, "Feature transformation for improved software bug detection models," in *ACM International Conference Proceeding Series*, Association for Computing Machinery, 2022. doi: 10.1145/3511430.3511444.

[22] B. Imran, E. Wahyudi, S. Riadi, Z. Muahidin, S. Erniwati, and W. A. Wahyuni, "A Comparative Hybrid Approach for Python Bug Detection Using Syntactic Features, Random Forest, and Neural Network," *CommIT J.*, vol. 19, no. 2, pp. 141–150, 2025.

[23] B. Imran, S. Riadi, E. Suryadi, M. Zulpahmi, and E. Wahyudi, "SemetonBug : A Machine Learning Model for Automatic Bug Detection in Python Code Based on Syntactic Analysis," *J. Inform.*, vol. 11, no. 2, pp. 75–80, 2025.

[24] H. M. Tran, S. T. Le, S. Van Nguyen, and P. T. Ho, "An Analysis of Software Bug Reports Using Machine Learning Techniques," *SN Comput. Sci.*, vol. 1, no. 1, 2020, doi: 10.1007/s42979-019-0004-1.

[25] W. Albattah and M. Alzahrani, "Software Defect Prediction based on Machine Learning and Deep Learning," *AI*, pp. 116–122, 2024, doi: 10.1109/ICICT54344.2022.9850643.