A Comparative Hybrid Approach for Python Bug Detection Using Syntactic Features, Random Forest, and Neural Network

Bahtiar Imran[1], Erfan Wahyudi[2*], Selamet Riadi[3], Zumratul Muahidin[4], Surni Erniwati[5], and Wenti Ayu Wahyuni[6]
[1,3]Computer Systems Engineering, Faculty of Information and Communication Technology, Universitas Teknologi Mataram
Nusa Tenggara Barat, Indonesia 83115
[2]Public Safety and Security Management, Faculty of Community Protection, Institut Pemerintahan Dalam Negeri (IPDN)
Jawa Barat, Indonesia 45363
[4]Information Systems, Faculty of Information and Communication Technology, Universitas Teknologi Mataram
Nusa Tenggara Barat, Indonesia 83115
[5]Information Management, Faculty of Vocational Studies, Universitas Teknologi Mataram
Nusa Tenggara Barat, Indonesia 83115
[6]Department of Informatics Engineering, Faculty of Information and Communication Technology, Universitas Teknologi Mataram
Nusa Tenggara Barat, Indonesia 83115
Email: [1]bahtiarimranlombok@gmail.com, [2]erfan.wahyudie@gmail.com, [3]didiriadijumantoro@gmail.com, [4]muahidinzumratul@gmail.com, [5]mentari1990@gmail.com, [6]wentiayu443322@gmail.com

***Abstract*—As software systems become increasingly complex, detecting bugs in source code has become a critical challenge in software development and maintenance. Manual debugging is time-consuming and error-prone, prompting the need for automated bug detection solutions. The research explores the use of machine learning models, specifically, Random Forest and Neural Network, for identifying bugs in Python source code. Features are extracted using Abstract Syntax Trees (ASTs), which enable the structured parsing of syntactic elements such as functions, classes, variables, conditionals, and exception blocks. These features serve as input to train both models for binary classification: distinguishing between buggy and non-buggy code files. Both buggy and non-buggy code files have 200 Python scripts. The models are evaluated using accuracy, confusion matrices, Receiver Operating Characteristic (ROC) curves, and classification reports across multiple training epochs. Experimental results show that the Random Forest model achieves stable performance with an accuracy of 86.67% and an Area Under the Curve (AUC) score of 0.97 on the test set, without significant improvement across epochs. In contrast, the Neural Network demonstrates gradual accuracy improvement from 68.33% at epoch 5 to 85% at epoch 300, along with higher sensitivity in bug detection, although it requires longer training times. Additionally, both models are used to predict specific lines of code containing potential bugs. Based on these findings, the choice of model depends on the application context. Random Forest offers faster deployment and consistent performance, while Neural Networks provide better adaptability to complex patterns and improved accuracy with sufficient training.**

***Index Terms*—Bug Detection, Syntactic Features, Random Forest, Neural Network**

Received: March 11, 2025; received in revised form: June 16, 2025; accepted: June 17, 2025; available online: Sep. 04, 2025.
*Corresponding Author

I. INTRODUCTION

TODAY'S increasingly complex software systems have made development and maintenance more demanding than ever. Developers devote significant time and effort to revising code, adding new features, identifying bugs, and fixing errors. Predictive models are increasingly used to automate error detection and improve overall efficiency to support these efforts [1–4]. However, software bugs remain a persistent chal-

lenge in the fast-paced development cycle, not only disrupting functionality and reliability but also leading to financial losses and reduced user satisfaction. As such, efficient bug detection methods have attracted considerable attention from researchers and practitioners alike [5].

Several approaches have been proposed to address the software bug prediction problem. Machine Learning (ML) approaches, in particular, have gained prominence due to their ability to learn patterns from historical data and predict defective modules using various metrics and computational techniques [6–10]. Several studies have explored different ML and Deep Learning (DL) methods for this purpose. The first example has empirically evaluated eight well-known algorithms for software bug prediction and found that DL models, particularly Long Short-Term Memory (LSTM), outperform others, achieving an accuracy of 87% [11]. However, this approach is limited to sequential data and does not exploit structural or syntactic features of source code, which can enhance detection performance. The second example has applied ML techniques to analyze bug reports and prioritize severity levels [12]. While it demonstrates promising results with Random Forest (75% accuracy), it lacks scalability for multi-file analysis and does not incorporate syntax-based feature extraction.

The third example has introduced an Adaptive Artificial Jellyfish Optimization algorithm combined with LSTM for software bug prediction. It achieves high accuracy (93.41% on the Promise dataset), outperforming traditional classifiers. Despite its success, the method relies heavily on hyperparameter tuning and requires extensive training times, making it inefficient for real-time applications. Furthermore, like many previous studies, it does not consider the structural characteristics of code, limiting its generalizability across programming languages. The fourth example has used natural language processing on DL features to identify software bugs, showing that combining feature selection, transfer learning, and ensemble techniques significantly improves prediction accuracy [13]. Meanwhile, the fifth example has applied feature transformation to raw software metrics, resulting in a 4.25% average improvement in recall values. However, it is based solely on textual or statistical features without leveraging the actual syntactic structure of code [14].

The DL has also been employed for bug prioritization. For instance, previous research has utilized RNN-LSTM networks on JIRA datasets, achieving 90% accuracy and improving the F-measure by up to 15.2% compared to KNN [15]. Another research has applied ensemble learning for bug report classification, reaching up to 96.72% accuracy with text augmentation [16]. Despite these advancements, most models still treat code as plain text or numerical data rather than exploiting its hierarchical structure. One recent approach has explored the use of Abstract Syntax Trees (ASTs) for code evaluation, highlighting their potential in capturing syntactic relationships within code [17]. However, it focuses primarily on code similarity and plagiarism detection rather than bug prediction.

The research addresses these limitations by proposing an automated bug detection framework that combines syntactic feature extraction using ASTs with a dual-model comparison: Random Forest for ML and Neural Network for DL. The approach systematically captures structural code features, such as the number of functions, classes, variables, conditionals, and exception blocks, which are often overlooked in existing studies [13, 14, 18]. Additionally, the system is designed to process multiple Python files simultaneously, offering a scalable solution suitable for large-scale software projects.

By leveraging both traditional and DL models, the researchers aim to provide a balanced perspective on performance trade-offs: speed and stability versus adaptability and accuracy. The dataset used consists of Python source code [19], allowing the researchers to evaluate the effectiveness of the method in real-world scenarios. The research contributes to the growing field of ML-based bug prediction by introducing a structured, scalable, and interpretable approach that improves upon current state-of-the-art methodologies.

II. Research Method

A. Data Collection and Processing

The data collection in the research is conducted manually by gathering Python code samples, which are then categorized into two distinct groups: buggy code and non-buggy code. The buggy code group consists of 200 Python scripts intentionally embedded with various types of errors, including syntax errors, logical flaws, and runtime exceptions. The non-buggy code group also contains 200 Python scripts that are verified to be error-free and executed without issues.

All code samples are carefully reviewed and validated to ensure accurate classification before being used in the research. The dataset is organized into two folders: "bugged" and "non_bugged". Each file in these directories is processed systematically using the extract_features_from_code() function to extract syntactic features such as the number of functions, classes, variables, lines of code, conditionals, and exception blocks. Following feature extraction, labeling is performed, where each sample is assigned a binary label: 1 for buggy code and 0 for non-buggy code [19].

This balanced dataset ensures equal representation from both categories, providing a solid foundation for training and evaluating the ML models used in the research.

B. Feature Extraction Using Abstract Syntax Tree (AST)

The AST module is utilized to analyze Python code. It also extracts specific features. The features extracted are as follows [17]:

1) The number of functions in the code (num_functions)
2) The number of classes in the code (num_classes),
3) The number of variable declarations (num_variables),
4) The number of lines of code (num_lines),
5) The number of conditional statements (if, while, for) (num_conditionals),
6) The number of try blocks used for exception handling (num_exceptions).

The formula used for this feature extraction is shown in Eq. (1). It has N as total number of nodes in the syntax tree, I as an indicator function that returns 1 if a node belongs to a specific type (e.g., function, class, variable) and 0 otherwise, and $node_n$ as the n-th node in the AST.

$$\text{Number of Features} = \sum_{(n=1)}^{N} I(\text{node}_n \, \epsilon \, \text{Specific Type}). \tag{1}$$

C. Model Training

The development of the Random Forest model is conducted using the sklearn library, which provides implementations for various ML algorithms, including Random Forest. The model is built using the RandomForestClassifier from sklearn.ensemble. Random Forest is an ensemble learning method that employs multiple decision trees to make predictions. Each tree in the forest predicts a label based on input features, and the final decision is determined through majority voting across all trees [19, 20]. In the research, the model is trained using a dataset with extracted and standardized features, which is achieved using the StandardScaler. The model is then trained using the fit() function with the training data (X_train and y_train) [20]. This training process enables the model to identify patterns within the data and learn to classify Python code as buggy or non-buggy. Formula used is Eq. (2). It has y as the final prediction of the model (bug detected or no bug detected), n as the number of trees in the forest, T_i as the prediction from the i-th tree based on the input x, and Majority Vote as the rule used to determine the final prediction by selecting the majority decision from all decision trees.

$$y = \text{Majority Vote}(\bigcup_{i=1}^{n} T_i(x)). \tag{2}$$

The development of the Neural Network model is carried out using the Keras library, which is part of TensorFlow. The model is constructed with an artificial Neural Network architecture comprising multiple layers [20]. The model is initialized as a Sequential model, meaning that layers are added in a logical order. The first layer consists of a Dense layer with 128 neurons and a Rectified Linear Unit (ReLU) activation function. It accepts input dimensions corresponding to the number of features in the data (X_train_scaled.shape) [1]. Following the first layer, a Dropout layer with a 50% dropout rate is added to prevent overfitting by randomly deactivating some neurons during training. The second layer is another Dense layer with 64 neurons and a ReLU activation function, followed by another Dropout layer [21]. The final output layer is a Dense layer with 1 neuron and a sigmoid activation function, producing an output value between 0 and 1, representing the class probability (bugged or non-bugged). The formula used is Eq. (3). It shows y as the output of the network (probability), σ as the sigmoid activation function, calculated in Eq. (4), w_i as the weight for the i-th feature, x_i as the value of the i-th feature, b as the bias term, and n as the number of features in the input.

$$y = \sigma(\sum_{i=1}^{n} w_i x_i + b), \tag{3}$$

$$\sigma(z) = \frac{1}{1 + e^{-z}}. \tag{4}$$

D. Evaluation Metrics

The evaluation of the models in the research is performed by calculating accuracy, the classification report, and the confusion matrix for both models (Random Forest and Neural Network) after training. Accuracy is determined by comparing the model's predictions with the test data and calculating the percentage of correct predictions. For the Random Forest model, the accuracy_score function from the sklearn.metrics library is used, while for the Neural Network model, the accuracy is calculated by comparing the predicted results (with a threshold of 0.5) against the actual labels [12]. The formula used is Eq. (5). It shows TP as true positives (correct predictions for the bugged class), TN as true negatives (correct predictions for the non-bugged class), FP as false positives (incorrect

predictions for the bugged class), and FN as false negatives (incorrect predictions for the non-bugged class).

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Samples}} = \frac{TP+TN}{TP+TN+FP+FN}. \quad (5)$$

Additionally, the classification report provides other metrics, such as precision, recall, and F1-score for each class (bugged and non-bugged). Precision measures the proportion of correct bugged class predictions out of all bugged class predictions, while recall assesses how many bugged instances were correctly detected by the model. The F1-score is the harmonic mean of precision and recall, offering a balanced view of the model's performance. The confusion matrix illustrates the distribution of correct and incorrect predictions in a 2×2 matrix format, which is used to evaluate the model's errors in greater detail. This matrix displays the number of True Positives, True Negatives, False Positives, and False Negatives. The Receiver Operating Characteristic (ROC) and Area Under the Curve (AUC) are also calculated to evaluate the model's quality from another perspective. The AUC reflects how well the model distinguishes between the two classes (bugged and non-bugged). A higher AUC indicates better model performance in predicting the correct class [22].

III. Results and Discussion

A. Bug Detection Implementation

The research is designed to detect whether a Python source code file contains bugs and to identify the specific lines where those bugs may occur. Two ML models, namely Random Forest and Neural Network, are employed for bug detection. The models are trained using features extracted from the source code syntax, such as the number of functions, classes, variables, and control structures, allowing the algorithms to learn patterns associated with bugs. The model performance is evaluated using a confusion matrix on the testing data. Only the testing data (30% of the dataset) is used for evaluation, as it is not involved in the training process, ensuring an objective assessment of the model's generalization ability. Both Random Forest and Neural Network demonstrate good classification accuracy and performance, although some prediction errors are observed in both models.

B. Model Performance

The bug detection results indicate that the Random Forest model exhibits reliable performance in capturing simple patterns based on extracted features. With high accuracy on the testing data, this model proves its effectiveness in more structured cases. On the other hand, the Neural Network demonstrates strong capability in recognizing more complex patterns. The use of dropout and class_weight adjustments helps the Neural Network mitigate data imbalance issues between the "bugged" and "non_bugged" classes. However, due to its increased complexity, the Neural Network requires a longer training time compared to the Random Forest model. Figure 1 presents an example of the model's bug detection results.

Figure 1 presents partial bug detection results across various source code files. It shows both models' predictions and the ground truth of bug presence. Each row displays the analyzed file, the model's prediction ("bug detected"), the actual bug status ("bugged"), and the specific line containing the bug. These results indicate that the model accurately detects bugs in all displayed samples, demonstrating strong performance in bug classification. However, the model's effectiveness is not solely determined by its ability to identify the existence of bugs but also by its precision in detecting their exact locations.

The performance of the Random Forest model in detecting bugs in Python source code is evaluated using a confusion matrix, as shown in Fig. 2. The matrix reveals that the model achieves strong classification accuracy with a balanced ability to distinguish between bugged and non-bugged files. Out of 31 actual buggy files, the model correctly identifies 28 as containing bugs (true positives), reflecting a high detection accuracy and strong sensitivity to bug-related code patterns. However, 3 buggy files are misclassified as non-buggy (false negatives), suggesting a low miss rate, which enhances the model's reliability for practical bug detection tasks. It suggests that the model rarely misses actual bugs, making it reliable for practical use. Conversely, among 29 non-buggy files, the model correctly classifies 24 as clean (true negatives), showcasing its effectiveness in avoiding false alerts. Nevertheless, 5 non-buggy files are incorrectly labeled as buggy (false positives), which may indicate challenges in distinguishing complex code patterns that resemble buggy syntax but are actually valid. While this number is relatively small, it highlights potential limitations in distinguishing certain syntactic structures that resemble buggy behavior but are not actually erroneous.

Overall, the Random Forest model achieves an accuracy of 86.67%, reflecting its robustness in bug detection tasks. The precision and recall values also indicate a well-balanced performance: the model not only identifies most true bugs effectively but also minimizes incorrect classifications. These findings suggest

```
filebug11.py | Predicted: Bug Detected | Actual: bugged - Bug Lines: [11, 13, 16, 18, 19]
filebug15.py | Predicted: Bug Detected | Actual: bugged - Bug Lines: [11, 12]
filebug (64).py | Predicted: Bug Detected | Actual: bugged - Bug Lines: [16, 17, 22, 25]
filebug  (87).py | Predicted: Bug Detected | Actual: bugged - Bug Lines: [11, 12, 13, 15, 17]
filebug (98).py | Predicted: Bug Detected | Actual: bugged - Bug Lines: [11, 14]
filebug (58).py | Predicted: Bug Detected | Actual: bugged - Bug Lines: [11, 14]
filebug (70).py | Predicted: Bug Detected | Actual: bugged - Bug Lines: [12, 14, 16]
filebug (71).py | Predicted: Bug Detected | Actual: bugged - Bug Lines: [11, 13, 16, 19, 20]
filebug5.py | Predicted: Bug Detected | Actual: bugged - Bug Lines: [11, 14, 15]
filebug (65).py | Predicted: Bug Detected | Actual: bugged - Bug Lines: [13, 14, 16]
filebug (76).py | Predicted: Bug Detected | Actual: bugged - Bug Lines: [14, 16, 17]
filebug4.py | Predicted: No Bug Detected | Actual: bugged - Bug Lines: [11, 14, 16, 17]
filebug10.py | Predicted: Bug Detected | Actual: bugged - Bug Lines: [12, 13]
filebug (38).py | Predicted: Bug Detected | Actual: bugged - Bug Lines: [15, 16, 20]
filebug (48).py | Predicted: Bug Detected | Actual: bugged - Bug Lines: [16, 18, 22]
filebug (75).py | Predicted: Bug Detected | Actual: bugged - Bug Lines: [13, 14, 16]
filebug (60).py | Predicted: Bug Detected | Actual: bugged - Bug Lines: [12, 14, 16]
filebug (67).py | Predicted: Bug Detected | Actual: bugged - Bug Lines: [11, 12, 13, 15, 17]
filebug (49).py | Predicted: Bug Detected | Actual: bugged - Bug Lines: [13, 14, 16]
filebug (66).py | Predicted: Bug Detected | Actual: bugged - Bug Lines: [14, 16, 17]
filebug (77).py | Predicted: Bug Detected | Actual: bugged - Bug Lines: [11, 12, 13, 15, 17]
filebug (73).py | Predicted: Bug Detected | Actual: bugged - Bug Lines: [11, 12, 14, 15, 16, 17, 19]
filebug (62).py | Predicted: Bug Detected | Actual: bugged - Bug Lines: [11, 13, 14, 20, 21, 22]
filebug(1).py | Predicted: Bug Detected | Actual: bugged - Bug Lines: [13, 14, 18]
```

Fig. 1. Sample output showing bug detection results for selected Python scripts, including predicted and actual bug status with corresponding line numbers.

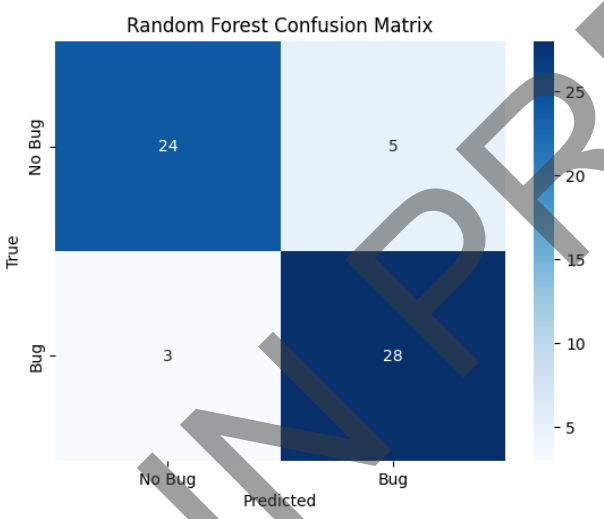

Fig. 2. Confusion matrix for Random Forest.

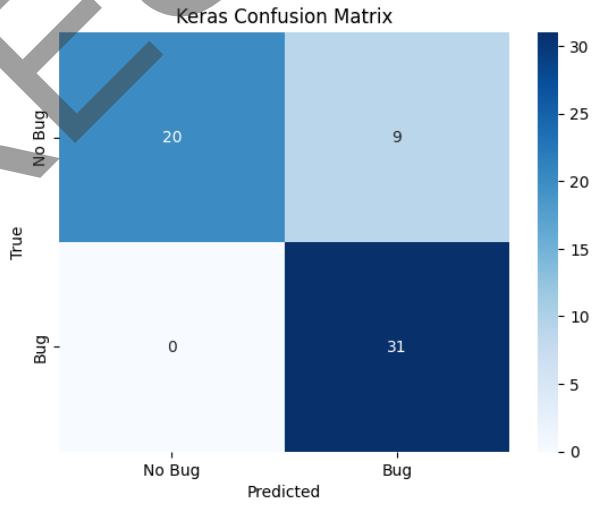

Fig. 3. Confusion matrix for Neural Network.

that the proposed method, leveraging syntactic features extracted via ASTs and trained using Random Forest, is both effective and efficient for automated bug detection in Python programs. Furthermore, the confusion matrix supports the conclusion that the model can be confidently deployed in early-stage software testing environments to help developers to identify potentially problematic code segments before deployment.

The Keras Neural Network model demonstrates strong performance in distinguishing between bugged and non-bugged Python files, as illustrated by the confusion matrix in Fig. 3. Out of 31 actual bugged files, the model correctly classifies all of them as containing bugs (true positives), resulting in zero false negatives. This outcome reflects perfect recall, indicating that the model does not miss any actual bugs, an essential quality in systems where undetected bugs may lead to critical failures. In contrast, among 29 non-bugged files, the model accurately identifies 20 files as non-bugged (true negatives) but misclassifies 9 files as bugged (false positives). Its relatively higher false positive rate suggests that the model adopts a conservative classification strategy, favoring bug detection at the

Cite this article as: 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 Journal 19(2), 141–150, 2025.

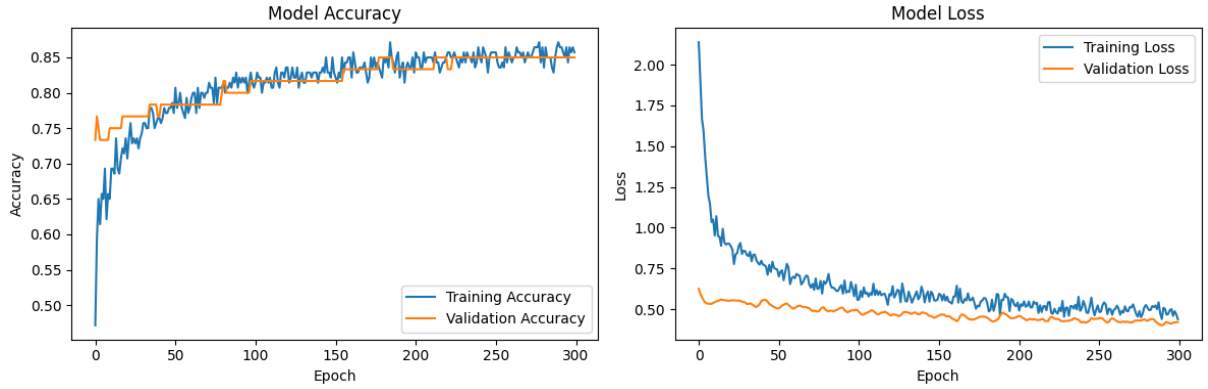

Fig. 4. Evaluation of Neural Network results.

expense of occasionally flagging clean code. While this cautious approach reduces the risk of undetected bugs, it may also result in unnecessary inspection of code that is actually correct.

The Keras Neural Network achieves an accuracy of 85% at epoch 300, demonstrating its capacity to learn meaningful patterns from syntactic features extracted via ASTs. Although this performance matches that of the Random Forest model in terms of accuracy, the neural network particularly excels in bug detection completeness, as evidenced by the absence of false negatives. This indicates strong sensitivity, making it suitable for applications where missing bugs could lead to critical issues.

However, compared to Random Forest, the Neural Network model produces a higher number of false positives, suggesting that further threshold tuning or post-processing may be necessary to enhance precision in real-world deployments. These results confirm the effectiveness of the proposed method, combining syntactic feature extraction and DL, for automated bug detection in Python programs. Despite requiring longer training times, the model's progressive accuracy improvement with increasing epochs underscores its potential for further optimization, especially in managing complex and large-scale software systems.

C. Evaluation of Detection Results

Figure 4 illustrates the training and validation performance of the Neural Network model over 300 epochs, using accuracy and loss metrics. The left plot shows a consistent upward trend in accuracy as training progresses, where both training and validation accuracy steadily improve and converge at approximately 85% by epoch 300. The validation accuracy closely follows the training accuracy with minor fluctuations, indicating good generalization capability and model stability across unseen data.

In the right plot, both training and validation loss experience a sharp decline during the initial epochs, particularly in the first 50, followed by a gradual convergence to lower and stable values. By epoch 300, the training loss is near 0.4, while the validation loss stabilizes around 0.45, with only a slight gap between the two. This small difference reflects minimal overfitting, suggesting that the model does not rely excessively on the training data and retains the ability to perform well on new, unseen inputs. Overall, the learning curves indicate that the model reaches its optimal performance after prolonged training, achieving a balance between bias and variance. The stable improvement across both accuracy and loss validates the model's effectiveness and reliability, making it well-suited for similar classification tasks involving syntactic features extracted from source code.

Figure 5 displays the ROC curve of the Random Forest model at epoch 300, which evaluates the classification performance based on the relationship between the True Positive Rate (TPR) and the False Positive Rate (FPR). The curve shows a steep rise towards the top-left corner, indicating excellent model performance. The Area Under the Curve (AUC) is 0.97, which reflects a very high ability of the model to distinguish between bugged and non-bugged files. This high AUC value suggests that the model achieves a strong trade-off between sensitivity and specificity, making it effective in correctly identifying buggy code while minimizing false alarms. The shape and position of the curve demonstrate that the Random Forest classifier performs consistently and generalizes well across the dataset. The near-perfect classification capability seen in this ROC curve confirms the robustness and reliability of the model when deployed in real-world software defect detection tasks. These results further support the findings from the confusion matrix and accuracy evaluations, showing that Random Forest

Cite this article as: 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 Journal 19(2), 141–150, 2025.

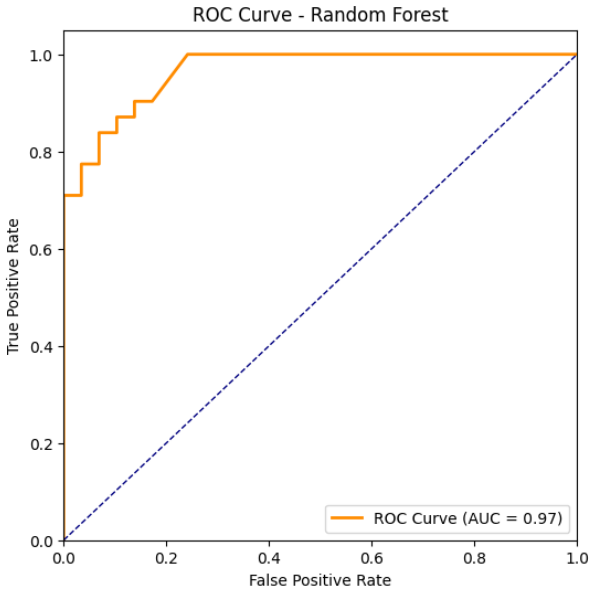

Fig. 5. Evaluation of Random Forest results.

TABLE I
RESULTS OF OVERALL EXPERIMENTS.

No	Epoch	Batch Size	Accuracy
1	5	32	Random Forest = 85.00% Neural Network = 68.33%
2	15	32	Random Forest = 85.00% Neural Network = 73.33%
3	25	32	Random Forest = 85.00% Neural Network = 75.00%
4	50	32	Random Forest = 85.00% Neural Network = 80.00%
5	72	32	Random Forest = 85.00% Neural Network = 81.67%
6	100	32	Random Forest = 85.00% Neural Network = 83.33%
7	200	32	Random Forest = 85.00% Neural Network = 83.33%
8	300	32	Random Forest = 86.67% Neural Network = 85.00%

remains a competitive method for bug prediction based on syntactic code features. The overall results from the conducted experiments can be seen in Table I.

Table I compares the accuracy of the Random Forest and Neural Network algorithms at various epoch levels, using a fixed batch size of 32. The Random Forest model consistently achieves an accuracy of 85.00% across most epochs, with a slight improvement to 86.67% at epoch 300, indicating its stability and robustness even without further training adjustments. In contrast, the Neural Network demonstrates a clear upward trend in accuracy as the number of epochs increases. Starting from 68.33% at epoch 5, the model gradually improves to 80.00% by epoch 50, and continues to rise, reaching 83.33% at epochs 100 and 200, and finally achieving 85.00% at epoch 300. This pattern reflects the Neural Network's ability to progressively learn and optimize its parameters through iterative training. The comparison highlights key differences between the two models. While Random Forest offers consistent and reliable performance with minimal tuning, the Neural Network shows greater adaptability and potential for improvement with extended training. However, the performance gain of the Neural Network starts to plateau after epoch 100, suggesting a convergence point. Interestingly, at epoch 300, the Neural Network catches up to match Random Forest's performance, indicating its competitive viability given sufficient training time. Thus, the choice between models depends on specific application needs: Random Forest is suitable for quick and stable deployment, whereas Neural Networks require more computational time but can achieve competitive results through deeper training.

D. Discussion

The research presents an analysis of experimental results alongside a comparative evaluation of the Random Forest and Neural Network models for bug detection in Python source code. The findings indicate that the Random Forest model consistently maintains stable accuracy across all testing scenarios, exhibiting minimal fluctuations regardless of the number of training epochs. In contrast, the Neural Network demonstrates a more dynamic learning trajectory, characterized by a gradual improvement in accuracy, ultimately reaching 85% at epoch 300. However, this enhanced performance comes at the cost of increased computational demands and longer training durations.

In terms of sensitivity, the Neural Network outperforms the Random Forest in detecting buggy code segments, albeit with a higher FPR, which can misclassify non-buggy code as buggy. It suggests that model selection should be based on specific project requirements. Random Forest is preferable in cases that prioritize speed and stability. At the same time, Neural Networks are more appropriate for applications that demand higher sensitivity and the ability to capture complex data patterns.

As shown in Table I, the Random Forest achieves an average accuracy of 85% consistently across different folds. Meanwhile, the Neural Network's accuracy improves progressively with training epochs. This trade-off performance emphasizes the balance between model complexity and efficiency. Evaluation using ROC curves confirms that both models possess strong classification capabilities, with the Neural Network achieving slightly higher AUC scores in most test cases.

Cite this article as: 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 Journal 19(2), 141–150, 2025.

Previous studies have explored a variety of ML and DL methods for bug detection, reflecting a growing interest in automated software quality assurance. Previous research has introduced BugLab, a self-supervised approach combining a detector and a selector model, achieving up to 30% improvement over baseline methods and discovering previously unreported bugs in Python open-source projects [23]. The NerdBug framework focuses on capturing neural network behaviors to detect bugs in DL systems [24]. A hybrid model based on Adaptive Artificial Jelly Optimization (A2JO) and LSTM networks has also been proposed for software bug prediction, demonstrating the versatility of evolutionary algorithms in ML pipelines [18]. Additionally, Theia has integrated dataset characteristics into bug localization tasks for Keras and PyTorch projects, showing how data profiles influence detection performance [25]. Last, Sydr-Fuzz, a fuzz testing pipeline, has been developed for robustness validation in Python-based ML frameworks [26].

In comparison with these previous studies, the research introduces a hybrid method that fuses rule-based syntactic parsing through ASTs with ML and DL classifiers. Unlike approaches that focus on single-model pipelines or specific bug types, the method enables multi-file analysis and balances the need for interpretability, accuracy, and computational efficiency. Future research may include statistical validation, such as confidence intervals and significance testing, to strengthen performance claims, as well as hyperparameter optimization and advanced DL architectures to further improve bug detection accuracy and scalability.

IV. Conclusion

The research demonstrates that both the Random Forest and Neural Network models possess valuable capabilities in detecting bugs within Python source code, each with distinct strengths. The Random Forest model exhibits consistently stable performance throughout training and achieves its highest accuracy of 86.67% at epoch 300, despite showing minimal variation across earlier epochs. This consistency makes it a reliable and efficient choice for scenarios requiring rapid deployment and dependable results. In contrast, the Neural Network model displays a progressive learning curve, gradually improving in accuracy and ultimately reaching 85% at epoch 300. This trend highlights its ability to capture more complex patterns in the data over time. Evaluations using confusion matrices and ROC curves further confirm that both models generalize well from training data, with the Neural Network demonstrating slightly higher sensitivity in identifying buggy code segments, albeit at the cost of a higher false positive rate.

The research findings hold meaningful implications for the field of automated software quality assurance. Integrating ML into static code analysis tools can serve as an effective initial screening mechanism before more resource-intensive dynamic testing is conducted. The proposed method can be implemented as a lightweight pre-screening tool during code reviews or within Continuous Integration (CI) and Continuous Delivery (CD) pipelines to help developers to identify potentially problematic code segments early in the development cycle.

Despite these promising outcomes, several limitations should be acknowledged. First, although the dataset enables preliminary analysis, it remains limited in scale and diversity, which may affect the generalizability of the findings. The performance of both models may also vary significantly when applied to larger, more diverse, or production-grade codebases due to the current dataset's synthetic nature and limited representativeness. Second, the feature extraction process is based on basic syntactic metrics such as the number of functions, classes, conditionals, and lines of code. While these features provide useful structural insights, they may not fully capture the semantic or contextual aspects that are critical for detecting more nuanced logical errors. Third, the bug line detection mechanism is rule-based and heuristic-driven, targeting a limited set of common error types, such as division by zero, missing else branches, and undeclared variable usage. It restricts its applicability to broader categories of real-world software defects.

Nevertheless, the research extends beyond binary classification by incorporating AST-based syntactic and semantic analysis to identify bug-prone lines within the source code. This rule-based approach operates independently of the training dataset, enhancing the system's interpretability and applicability in real-world development environments. In future research, the researchers aim to expand the dataset using real-world open-source repositories and leverage automated bug injection techniques to increase diversity and robustness.

Moreover, future researchers should aim to enhance the system's capabilities by incorporating context-aware and semantic-level code representations. Potential approaches include leveraging AST embeddings, code property graphs, or data flow analysis to capture program behavior better. Additionally, exploring advanced DL architectures, such as Graph Neural Networks (GNNs) or transformer-based models like CodeBERT and GraphCodeBERT, can significantly improve prediction accuracy and cross-project generalization. By bridging syntactic analysis and ML, the researchers provide a practical and scalable foundation

for integrating intelligent bug detection into modern software development workflows.

Author Contribution

Developed the conceptual idea, determined the analytical methods, and designed the model, B. I.; Conducted data preprocessing, executed the analysis, and interpreted the results, B. I.; Became responsible for composing the manuscript, B. I.; Gathered the dataset through field observation and manual recording, E. W.; Provided support in code analysis and debugging during the evaluation process, E. W.; Provided support in proofreading and refining the manuscript, E. W.; Shared relevant datasets and preprocessing tools used in the analysis, S. R. and Z. M.; Classified and organized buggy code according to predefined categories, Z. M.; Provided support in organizing and composing the manuscript, S. E.; Classified and organized buggy and non-buggy code segments according to their characteristics to support further analysis, S. E.; Supported the analysis of source code and participated in generating visual representations of the findings, W. A. W.; Conducted a literature search to identify relevant studies supporting the research, W. A. W.

Data Availability

The data that support the findings of the research are available from the corresponding author, Erfan Wahyudi, upon reasonable request.

References

[1] S. Mostafa, S. T. Cynthia, B. Roy, and D. Mondal, "Feature transformation for improved software bug detection and commit classification," *Journal of Systems and Software*, vol. 219, pp. 1–13, 2025.

[2] G. Giray, K. E. Bennin, Ö. Köksal, Ö. Babur, and B. Tekinerdogan, "On the use of deep learning in software defect prediction," *The Journal of Systems & Software*, vol. 195, pp. 1–26, 2023.

[3] J. A. Fadhil, K. T. Wei, and K. S. Na, "Artificial intelligence for software engineering: An initial review on software bug detection and prediction," *Journal of Computer Science*, vol. 16, no. 12, pp. 1709–1717, 2020.

[4] K. Krishna, P. Murthy, and S. Sarangi, "Exploring the synergy between generative AI and software engineering: Automating code optimization and bug fixing," vol. 13, no. 1, pp. 682–691, 2024.

[5] M. Nevendra and P. Singh, "Software defect prediction using deep learning," *Acta Polytechnica Hungarica*, vol. 18, no. 10, pp. 173–189, 2021.

[6] A. Khalid, G. Badshah, N. Ayub, M. Shiraz, and M. Ghouse, "Software defect prediction analysis using machine learning techniques," *Sustainability*, vol. 15, no. 6, pp. 1–17, 2023.

[7] A. Hammouri, M. Hammad, M. Alnabhan, and F. Alsarayrah, "Software bug prediction using machine learning approach," *International Journal of Advanced Computer Science and Applications*, vol. 9, no. 2, pp. 78–83, 2018.

[8] T. Hai, J. Zhou, N. Li, S. K. Jain, S. Agrawal, and I. B. Dhaou, "Cloud-based bug tracking software defects analysis using deep learning," *Journal of Cloud Computing*, vol. 11, no. 1, pp. 1–14, 2022.

[9] N. Tabassum, A. Namoun, T. Alyas, A. Tufail, M. Taqi, and K. H. Kim, "Classification of bugs in cloud computing applications using machine learning techniques," *Applied Sciences*, vol. 13, no. 5, pp. 1–24, 2023.

[10] S. D. Immaculate, M. F. Begam, and M. Floramary, "Software bug prediction using supervised machine learning algorithms," in *2019 International conference on data science and communication (IconDSC)*. Bangalore, India: IEEE, March 1–2, 2019, pp. 1–7.

[11] W. Albattah and M. Alzahrani, "Software defect prediction based on machine learning and deep learning techniques: An empirical approach," *AI*, vol. 5, no. 4, pp. 1743–1758, 2024.

[12] H. M. Tran, S. T. Le, S. V. Nguyen, and P. T. Ho, "An analysis of software bug reports using machine learning techniques," *SN Computer Science*, vol. 1, 2020.

[13] R. Siva, S. Kaliraj, B. Hariharan, and N. Premkumar, "Automatic software bug prediction using adaptive artificial jelly optimization with Long Short-Term Memory," *Wireless Personal Communications*, vol. 132, no. 3, pp. 1975–1998, 2023.

[14] Q. M. Ul Haq, F. Arif, K. Aurangzeb, N. Ul Ain, J. A. Khan, S. Rubab, and M. S. Anwar, "Identification of software bugs by analyzing natural language-based requirements using optimized deep learning features," *Computers, Materials & Continua*, vol. 78, no. 3, pp. 4379–4397, 2024.

[15] S. T. Cynthia, B. Roy, and D. Mondal, "Feature transformation for improved software bug detection models," in *Proceedings of the 15th Innovations in Software Engineering Conference*. Gandhinagar, India: Association for Computing Machinery, Feb. 24–26, 2022, pp. 1–10.

[16] H. Bani-Salameh, M. Sallam, and B. Al Shboul, "A deep-learning-based bug priority prediction using RNN-LSTM neural networks," *E-Informatica Software Engineering Journal*,

vol. 15, no. 1, pp. 29–45, 2021.

[17] S. A. Alsaedi, A. Y. Noaman, A. A. A. Gad-Elrab, and F. E. Eassa, "Nature-based prediction model of bug reports based on ensemble machine learning model," *IEEE Access*, vol. 11, pp. 63 916–63 931, 2023.

[18] A. T. P. Nguyen and V. D. Hoang, "Development of code evaluation system based on abstract syntax tree," *Journal of Technical Education Science*, vol. 19, no. Special Issue 01, pp. 15–24, 2024.

[19] P. Batchu, T. R. Gali, and S. Inturi, "Python source code analysis for bug detection using transformers," *Engineering Technology Journal*, vol. 09, no. 04, pp. 3772–3777, 2024.

[20] G. Giray, "A software engineering perspective on engineering machine learning systems: State of the art and challenges," *Journal of Systems and Software*, vol. 180, 2021.

[21] 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. 200 749–200 763, 2020.

[22] L. Gomes, R. da Silva Torres, and M. L. Côrtes, "BERT-and TF-IDF-based feature extraction for long-lived bug prediction in FLOSS: A comparative study," *Information and Software Technology*, vol. 160, 2023.

[23] M. Allamanis, H. Jackson-Flux, and M. Brockschmidt, "Self-supervised bug detection and repair," *Advances in Neural Information Processing Systems*, vol. 34, pp. 27 865–27 876, 2021.

[24] F. Jafarinejad, K. Narasimhan, and M. Mezini, "Nerdbug: Automated bug detection in Neural Networks," in *Proceedings of the 1st ACM International Workshop on AI and Software Testing/Analysis*. Virtual, Denmark: Association for Computing Machinery, July 12, 2021, pp. 13–16.

[25] R. Manke, M. Wardat, F. Khomh, and H. Rajan, "Leveraging data characteristics for bug localization in deep learning programs," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 6, pp. 1–29, 2025.

[26] I. Yegorov, E. Kobrin, D. Parygina, A. Vishnyakov, and A. Fedotov, "Python fuzzing for trustworthy machine learning frameworks," *Journal of Mathematical Sciences*, vol. 285, no. 2, pp. 180–188, 2024.