JURNAL INFORMATIKA, Vol. 11 No. 2 October 2025, Page 75-80
ISSN (Print) 2355-6579 | ISSN (Online) 2528-2247
DOI: https://doi.org/10.31294/inf.v12i2.25340

SemetonBug: A Machine Learning Model for Automatic Bug Detection in Python Code Based on Syntactic Analysis

Bahtiar Imran[1], Selamet Riadi[2*], Emi Suryadi[3], M. Zulpahmi[4], Zaeniah[5], Erfan Wahyudi[6]

[1,2,3]Rekayasa Sistem Komputer, Fakultas Teknologi Informasi dan Komunikasi, Universitas Teknologi Mataram, Mataram, Indonesia
[4]Teknik Informatika, Fakultas Teknologi Informasi dan Komunikasi, Universitas Teknologi Mataram, Mataram, Indonesia
[5]Sistem Informasi, Fakultas Teknologi Informasi dan Komunikasi, Universitas Teknologi Mataram, Mataram, Indonesia
[6]Manajemen Keamanan dan Keselamatan Publik, Institut Pemerintahan Dalam Negeri, Praya, Indonesia

Correspondence e-mail: didiriadijumantoro@gmail.com

Submission:	Revision:	Acceptance:	Available Online:
22-02-2025	19-04-2025	19-06-2025	01-10-2025

Abstract
Bug detection in Python programming is a crucial aspect of software development. This study develops an automated bug detection system using feature extraction based on Abstract Syntax Tree (AST) and a Random Forest Classifier model. The dataset consists of 100 manually classified bugged files and 100 non-bugged files. The model is trained using structural code features such as the number of functions, classes, variables, conditions, and exception handling. Evaluation results indicate an accuracy of 86.67%, with balanced precision and recall across both classes. Confusion matrix analysis identifies the presence of false positives and false negatives, albeit in relatively low numbers. The accuracy curve suggests a potential overfitting issue, as training accuracy is higher than testing accuracy. This study demonstrates that the combination of AST-based feature extraction and Random Forest can be an effective approach for automated bug detection, with potential improvements through model optimization and a larger dataset.

Keywords: Abstract Syntax Tree, Random Forest, Machine learning.

1. Introduction

Bug detection in Python programming is a crucial aspect of developing high-quality software (Adarsh et al., 2023; Albattah & Alzahrani, 2024; Hammouri et al., 2018; Immaculate et al., 2022; Verma, 2024). Bugs can arise from various sources, including syntax errors, logic errors, or even improper use of Python's dynamic features. Research has shown that Python programming has unique characteristics that influence the patterns of bugs that emerge. For instance, an empirical study conducted by Hu and Zhang found that numerous previously undetected bugs exist in commonly used Python libraries such as Pillow, highlighting the need for more effective bug detection tools (Hu & Zhang, 2022). Additionally, Chen et al. revealed that dynamic code changes during bug fixing significantly impact software quality, making it essential to understand how these features interact when addressing bugs (Chen et al., 2017).

In this context, the development of tools and techniques for bug detection becomes highly critical. For example, BugsInPy is a database designed to support research and development of testing and debugging tools specifically for Python programs, reducing barriers to research in this field (Widyasari et al., 2020).
Moreover, a bug prediction model developed by Khan et al. demonstrated that hyperparameter optimization in machine learning algorithms can improve the accuracy of software bug prediction (Khan et al., 2020). Furthermore, research by Hammouri et al. indicated that applying machine learning algorithms such as Naïve Bayes, Decision Trees, and Artificial Neural Networks yields better results in bug detection compared to conventional methods (Hammouri et al., 2018).
Several prior studies have focused on Python bug detection : (Zhang et al., 2014) developed a system called AI designed to address concurrent bugs. Although the system successfully detected many bugs, it failed to identify specific synthetic bugs requiring interleaving access to shared variables by two threads, highlighting limitations in its approach. (Elmishali et al., 2019) introduced

DeBGUer, a tool employing the Learn, Diagnose, and Plan (LDP) paradigm to detect and isolate bugs. The error prediction model trained with machine learning effectively guided testing efforts, demonstrating significant potential in enhancing bug detection efficiency. (Alam Zaidi et al., 2020) applied convolutional neural networks (CNN) to recommend bug fixes. Experimental results showed that the ELMo-CNN-based approach achieved the highest accuracy in bug triage problems, highlighting the effectiveness of deep learning techniques in addressing bug-related challenges in large software projects. (Allamanis et al., 2021) developed the BUGLAB model, which uses self-supervised learning to detect and fix bugs. This model improved detection accuracy by up to 30% compared to baseline methods and discovered 19 previously undetected bugs in open-source software. (Deng et al., 2024) conducted a testing campaign to identify logic bugs in spatial database engines using a geometry-aware generator. Their research successfully detected 34 unique bugs, 30 of which were confirmed, and 18 were fixed, demonstrating the effectiveness of their technique in uncovering bugs overlooked by previous methodologies. Lastly. (Shukla et al., 2021) designed an automated approach to detect, localize, and fix bugs in P4 programs using machine learning guided by fuzzing. Their approach successfully detected runtime bugs without modifying P4 programs, indicating great potential in automating the debugging process.

This study aims to develop an automated bug detection system in Python code using syntactic analysis and machine learning with the Random Forest model. This method extracts syntactic features from source code using Abstract Syntax Tree (AST) without requiring direct execution, making it safer and more efficient than traditional execution-based or rule-based methods. The dataset is collected directly, consisting of 100 bugged files and 100 non-bugged files, manually classified. The model is trained using features such as the number of functions, classes, variables, conditions, and exception handling blocks to identify patterns potentially containing bugs. The novelty of this research lies in the combination of AST-based feature extraction and Random Forest Classifier for bug detection, an approach that has been rarely explored. The results of this study are expected to enhance debugging efficiency and contribute to the automation of code review as well as Continuous Integration and Continuous Deployment (CI/CD) pipelines.

2. Research Methods

2.1. Data Collection

In this study, data was collected directly to ensure the diversity and quality of the dataset. The dataset consists of 100 bugged files and 100 non-bugged files, organized into two separate directories: "bugged" for code containing bugs and "non_bugged" for bug-free code. Each Python file was analyzed using Abstract Syntax Tree (AST) to extract structural features, such as the number of functions, classes, variables, conditions, and exception handling blocks. The extracted data was then converted into feature vectors and labeled as 1 for bugged code and 0 for non-bugged code. This approach ensures that the dataset reflects variations in code structure and complexity levels, making it suitable for training a machine learning model to detect bugs automatically.

2.2. Feature Extraction Using AST

In this study, feature extraction was performed using Abstract Syntax Tree (AST) to analyze the syntactic structure of Python code. AST allows for breaking down the source code into its fundamental components, such as functions, classes, variables, conditions, and exception handling. Several structural features were extracted from each Python file, including the number of functions (FunctionDef), classes (ClassDef), declared variables (Assign), total lines of code, branching and looping structures (If, While, For), as well as exception handling mechanisms (Try). This approach enables the machine learning model to recognize structural patterns in the code and differentiate between bugged and non-bugged files. If a SyntaxError is encountered during AST parsing, the file is skipped to ensure the training process remains uninterrupted. (Nguyen & Hoang, 2024).

2.3. Formulas Used

The following formulas were applied to extract structural features from the Python code using Abstract Syntax Tree (AST):

Number of functions: $F = \sum FunctionDef$
Number of classes: $C = \sum ClassDef$
Number of declared variables: $V = \sum Assign$
Total lines of code: $L = total\ lines$
Number of conditions (if, for, while): $Cond = \sum(If, For, While)$
Number of exception handling blocks: $E = \sum Try$

Each code is represented as a feature vector.

$$X = [F, C, V, L, Cond, E] \quad (1)$$

2.4. Training the Model with Random Forest

After extracting features from the dataset, the Random Forest Classifier model is used to detect bugs in Python code (P & Kambli, 2020).

The dataset, consisting of 100 bugged files and 100 non-bugged files, is split into training data (70%) and testing data (30%) using a train-test split. The model is trained with 100 decision trees (estimators) to classify whether a given code contains a bug or not. During the training process, the model learns patterns from the structural features of the code extracted using AST. After training, the model is evaluated using accuracy, a confusion matrix, and a classification report. Additionally, the accuracy curve for both training and testing is plotted to analyze model performance as the number of estimators increases.
The formulas used in this study are as follows:

$$Entropy = -\sum_{i=1}^{n} pi \; log \; 2pi \quad (2)$$

Where pi is the probability of a sample belonging to a certain category.
The model is trained with nnn estimators (number of trees).

$$model = RandomForestClassifier(n_{estimators} = 100, random_{state} = 42) \quad (3)$$

2.5. Model Evaluation

After training, the Random Forest Classifier model is evaluated to measure its performance in detecting bugs in Python code. The evaluation is conducted using accuracy, a classification report, and a confusion matrix. Accuracy is calculated by comparing the model's predictions with the actual labels in the test data. Additionally, the classification report provides metrics such as precision, recall, and F1-score to assess how well the model classifies bugged and non-bugged code. The confusion matrix is visualized as a heatmap to show the number of correct and incorrect predictions (Meenakshi & Singh, 2019). Furthermore, training and testing accuracy curves are plotted based on the number of estimators to analyze potential overfitting or underfitting, ensuring that the model can generalize well to new data (Mostafa et al., 2025).

Formulas Used:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Samples}} = \frac{TP + TN}{TP + TN + FP + FN} \quad (4)$$

2.6. Prediction and Implementation

After the Random Forest Classifier model has been trained and evaluated, the next step is to utilize it for automatic bug detection in Python code. The model predicts whether a Python file contains bugs by extracting structural features using the Abstract Syntax Tree (AST) and comparing them with patterns learned during training. The prediction process can be applied to individual files or an entire directory containing categorized bugged and non-bugged code. The prediction results are displayed alongside the actual labels for further validation.

The trained model is employed to detect bugs in new code by transforming the code into a feature vector $\boldsymbol{X}$ and predicting with:

$$\boldsymbol{y = model.predict(X)} \quad (5)$$

3. Results and Discussion

3.1. Bug Detection Implementation

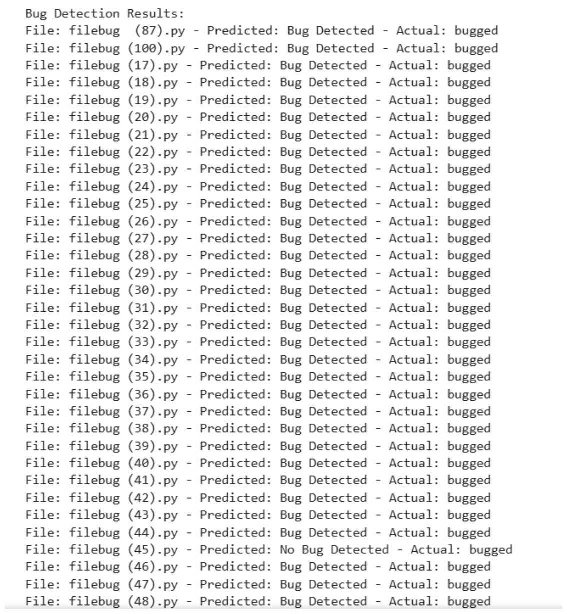
After the Random Forest Classifier model has been trained and evaluated, the bug detection implementation is conducted by testing the model on Python code files categorized as bugged and non-bugged. This process involves reading each file in the test directory, extracting features using the Abstract Syntax Tree (AST), and predicting whether the file contains bugs.

The prediction results are compared with the actual labels to assess the model's effectiveness. Based on the evaluation, the model demonstrates a high accuracy in detecting bugs, as indicated by the classification report and confusion matrix. The model performs well in classifying bugged and non-bugged code, with balanced precision and recall, ensuring that it is not only accurate in detecting bugs but also minimizes misclassification of correct code as bugged (false positives).

Furthermore, the analysis of training and testing accuracy curves indicates that the model does not suffer from overfitting, as its performance on test data remains stable with an increasing number of estimators in the Random Forest. This suggests that the model has good generalization capability for new data.

3.2. Proposed Model Performance

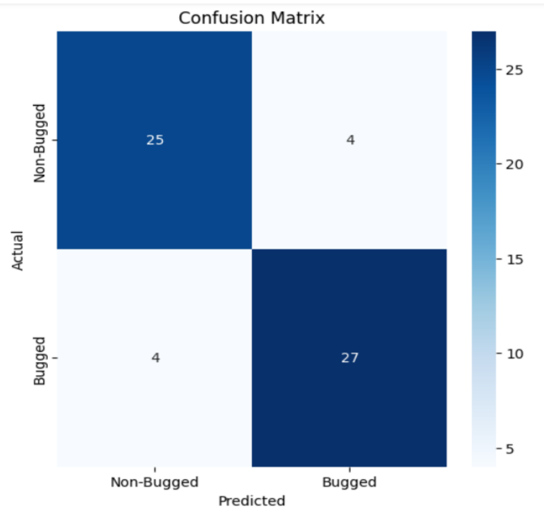
The Random Forest Classifier developed in this study demonstrates a satisfactory performance in detecting bugs in Python code, achieving an accuracy rate of 86.67%. This accuracy reflects the model's capability to correctly classify source code into bugged and non-bugged categories. Beyond accuracy, the classification report provides additional insights into the balance between precision, recall, and F1-score for each class. For the non-bugged class (label 0), the model achieves a precision of 0.86, recall of 0.86, and an F1-score of 0.86 from a total of 29 samples. Meanwhile, for the bugged class (label 1), the model records a precision of 0.87, recall of 0.87, and an F1-score of 0.87 from a total of 31 samples. The high precision values for both classes indicate that the model effectively minimizes false positives when identifying bugged code. Additionally, the high recall values suggest that the model successfully detects most of the actual bugged code. With balanced F1-scores, the model demonstrates a low error rate and stable performance in bug detection.



```
Bug Detection Results:
File: filebug  (87).py - Predicted: Bug Detected - Actual: bugged
File: filebug (100).py - Predicted: Bug Detected - Actual: bugged
File: filebug (17).py - Predicted: Bug Detected - Actual: bugged
File: filebug (18).py - Predicted: Bug Detected - Actual: bugged
File: filebug (19).py - Predicted: Bug Detected - Actual: bugged
File: filebug (20).py - Predicted: Bug Detected - Actual: bugged
File: filebug (21).py - Predicted: Bug Detected - Actual: bugged
File: filebug (22).py - Predicted: Bug Detected - Actual: bugged
File: filebug (23).py - Predicted: Bug Detected - Actual: bugged
File: filebug (24).py - Predicted: Bug Detected - Actual: bugged
File: filebug (25).py - Predicted: Bug Detected - Actual: bugged
File: filebug (26).py - Predicted: Bug Detected - Actual: bugged
File: filebug (27).py - Predicted: Bug Detected - Actual: bugged
File: filebug (28).py - Predicted: Bug Detected - Actual: bugged
File: filebug (29).py - Predicted: Bug Detected - Actual: bugged
File: filebug (30).py - Predicted: Bug Detected - Actual: bugged
File: filebug (31).py - Predicted: Bug Detected - Actual: bugged
File: filebug (32).py - Predicted: Bug Detected - Actual: bugged
File: filebug (33).py - Predicted: Bug Detected - Actual: bugged
File: filebug (34).py - Predicted: Bug Detected - Actual: bugged
File: filebug (35).py - Predicted: Bug Detected - Actual: bugged
File: filebug (36).py - Predicted: Bug Detected - Actual: bugged
File: filebug (37).py - Predicted: Bug Detected - Actual: bugged
File: filebug (38).py - Predicted: Bug Detected - Actual: bugged
File: filebug (39).py - Predicted: Bug Detected - Actual: bugged
File: filebug (40).py - Predicted: Bug Detected - Actual: bugged
File: filebug (41).py - Predicted: Bug Detected - Actual: bugged
File: filebug (42).py - Predicted: Bug Detected - Actual: bugged
File: filebug (43).py - Predicted: Bug Detected - Actual: bugged
File: filebug (44).py - Predicted: Bug Detected - Actual: bugged
File: filebug (45).py - Predicted: No Bug Detected - Actual: bugged
File: filebug (46).py - Predicted: Bug Detected - Actual: bugged
File: filebug (47).py - Predicted: Bug Detected - Actual: bugged
File: filebug (48).py - Predicted: Bug Detected - Actual: bugged
```

Source : Research process
Figure 1. Example of Bug Detection Results

Figure 1 illustrates the bug detection results, demonstrating that the model successfully identified most files containing bugs correctly. From the list of results, nearly all files named "filebug ().py" were detected as "Bug Detected," aligning with their actual labels as "bugged." However, there was one misclassification case, "filebug (45).py," where the model predicted "No Bug Detected," despite the file actually containing a bug. This indicates a potential false negative in the classification. Overall, the model performed well in detecting bugs in Python code, with only a few misclassification errors.

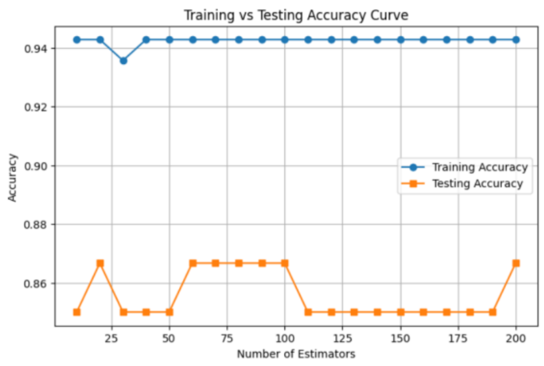


Source : Research process
Figure 2. Confusion Matrix

Figure 2 illustrates the model's performance in detecting both bugged and non-bugged code. The model correctly classified 27 bugged samples as bugged (True Positive) and 25 non-bugged samples as non-bugged (True Negative). However, there were 4 False Positive cases, where non-bugged code was incorrectly classified as bugged, and 4 False Negative cases, where bugged code was misclassified as non-bugged. These results indicate that the model maintains a good balance in distinguishing between bugged and non-bugged code, with a relatively low error rate.

3.1. Evaluation of Detection Results

The evaluation of bug detection results indicates that the model performs well in classifying code as bugged or non-bugged. Most files containing bugs were correctly identified, aligning with their actual labels. However, some misclassifications occurred, such as with *filebug (45).py*, where the model incorrectly classified a bugged file as non-bugged. This error suggests the presence of false negatives, which could lead to undetected bugs in the code. Despite this, with an accuracy of 86.67%, the model still demonstrates reliable performance in detecting bugs in Python code. Further evaluation can be conducted by increasing the training dataset size or optimizing the model's parameters to reduce misclassification errors.



Source : Research process
Figure 3. Training and Testing Accuracy Curve

Figure 3 illustrates the accuracy curve of the Random Forest model based on the number of estimators. The blue line represents training accuracy, which remains stable at approximately 94%, while the orange line represents testing accuracy, which hovers around 86.67%. The graph indicates that the training accuracy is significantly higher than the testing accuracy, suggesting potential overfitting. This means the model has learned patterns too specific to the training data, limiting its ability to generalize to new data.

Additionally, as the number of estimators increases, testing accuracy tends to plateau, implying that adding more estimators does not necessarily improve the model's performance. The achieved testing accuracy of 86.67% demonstrates that the model is fairly reliable in detecting bugs in Python code. However, there is still room for improvement to reduce the gap between training and testing accuracy. Furthermore, after reaching approximately 75 estimators, testing accuracy stagnates, indicating that increasing the number of estimators beyond a certain point does not provide substantial performance gains. This highlights the importance of selecting an optimal number of estimators to prevent excessive model complexity without significant improvement in testing performance.

4. Conclusion

This study successfully developed an automatic bug detection model for Python code using Abstract Syntax Tree (AST) and Random Forest Classifier. The dataset consisted of 100 bugged files and 100 non-bugged files, with the model trained using structural code features such as the number of functions, classes, variables, conditions, and exception handling.
Evaluation results indicate that the model achieved an accuracy of 86.67%, with balanced precision and recall for both classes, demonstrating reliable bug detection performance. The confusion matrix revealed four False Positive cases and four False Negative cases, indicating that while the model is fairly accurate, some classification errors remain.

Analysis of the accuracy curve suggests potential overfitting, as the training accuracy reached 94%, while the testing accuracy remained at 86.67%. This indicates that the model is overly fitted to the training data, limiting its performance on new data. Furthermore, increasing the number of estimators beyond 75 did not yield a significant improvement in accuracy, highlighting the importance of optimal parameter selection.

Overall, this study demonstrates that the AST-based feature extraction approach combined with the Random Forest classifier can enhance the efficiency of bug detection in Python code. However, to further improve accuracy, model optimization and dataset expansion are necessary to reduce classification errors and enhance the model’s generalization capability.

Reference

Adarsh, T. K., Sinchana, R., C, K. P., & Uday, R. (2023). Software Bug Prediction Using Machine Learning Approach. *International Journal for Research in Applied Science & Engineering Technology (IJRASET)*, *11*(Xii), 1401–1405.

Alam Zaidi, S. F., Awan, F. M., Lee, M., Woo, H., & Lee, C.-G. (2020). Applying Convolutional Neural Networks With Different Word Representation Techniques to Recommend Bug Fixers. *Ieee Access*. https://doi.org/10.1109/access.2020.3040065

Albattah, W., & Alzahrani, M. (2024). Software Defect Prediction based on Machine Learning and Deep Learning. *AI*, 116–122. https://doi.org/10.1109/ICICT54344.2022.9850643

Allamanis, M., Jackson-Flux, H., & Brockschmidt, M. (2021). *Self-Supervised Bug Detection and Repair*. https://doi.org/10.48550/arxiv.2105.12787

Chen, Z., Ma, W., Wei, L., Chen, L., Li, Y., & Xu, B. (2017). A Study on the Changes of Dynamic Feature Code When Fixing Bugs: Towards the Benefits and Costs of Python Dynamic Features. *Science China Information Sciences*. https://doi.org/10.1007/s11432-017-9153-3

Deng, W., Mang, Q., Zhang, C., & Rigger, M. (2024). Finding Logic Bugs in Spatial Database Engines Via <i>Affine Equivalent Inputs</I>. *Proceedings of the Acm on Management of Data*. https://doi.org/10.1145/3698810

Elmishali, A., Stern, R., & Kalech, M. (2019). DeBGUer: A Tool for Bug Prediction and Diagnosis. *Proceedings of the Aaai Conference on Artificial Intelligence*. https://doi.org/10.1609/aaai.v33i01.33019446

Hammouri, A., Hammad, M., Alnabhan, M. M., Alnabhan, M., & Alsarayrah, F. (2018). Software Bug Prediction using Machine Learning Approach Network Routing View project E-learning View project Software Bug Prediction using Machine Learning Approach. *Article in International Journal of Advanced Computer Science and Applications*, *9*(2), 78–83. www.ijacsa.thesai.org

Hu, M., & Zhang, Y. (2022). An Empirical Study of the Python/C API on Evolution and Bug Patterns. *Journal of Software Evolution and Process*. https://doi.org/10.1002/smr.2507

Immaculate, S. D., Begam, M. F., & Floramary< M. (2022). Software Bug Prediction Using Supervised Machine Learning Algorithms. *IEEE Access*, 849–869. https://doi.org/10.4018/978-1-6684-6291-1.ch044

Khan, F., Kanwal, S., Alamri, S., & Mumtaz, B. (2020). Hyper-Parameter Optimization of Classifiers, Using an Artificial Immune Network and Its Application to Software Bug Prediction. *Ieee Access*.

https://doi.org/10.1109/access.2020.2968362

Meenakshi, & Singh, D. S. (2019). Software Bug Prediction Using Supervised Machine Learning Algorithms. *International Research Journal of Engineering and Technology (IRJET)*, 4968–4971. https://doi.org/10.1109/IconDSC.2019.8816965

Mostafa, S., Cynthia, S. T., Roy, B., & Mondal, D. (2025). Feature transformation for improved software bug detection and commit classification. *Journal of Systems and Software*, *219*(July 2024), 112205. https://doi.org/10.1016/j.jss.2024.112205

Nguyen, A.-T. P., & Hoang, V.-D. (2024). Development of Code Evaluation System based on Abstract Syntax Tree. *Journal of Technical Education Science*, *19*(1), 15–24. https://doi.org/10.54644/jte.2024.1514

P, R., & Kambli, P. (2020). Analysis on Detecting a Bug in a Software using Machine Learning. *International Journal of Recent Technology and Engineering (IJRTE)*, *9*(2), 1195–1199. https://doi.org/10.35940/ijrte.b4119.079220

Shukla, A., Hudemann, K. N., Vági, Z., Hügerich, L., Smaragdakis, G., Hecker, A., Schmid, S., & Feldmann, A. (2021). *Fix With P6: Verifying Programmable Switches at Runtime*. https://doi.org/10.1109/infocom42981.2021.9488772

Verma, K. (2024). Bug Prediction using Machine Learning. *International Journal of Computer Science & Communication*, *15*(1), 15–23.

Widyasari, R., Sim, S. Q., Lok, C., Qi, H., Phan, J., Tay, Q., Tan, C., Wee, F., Tan, J. E., Yieh, Y., P. Goh, B. K., Thung, F., Kang, H. J., Hoang, T., Lo, D., & Ouh, E. L. (2020). *BugsInPy: A Database of Existing Bugs in Python Programs to Enable Controlled Testing and Debugging Studies*. https://doi.org/10.1145/3368089.3417943

Zhang, M., Wu, Y., Lu, S., Qi, S., Ren, J., & Zheng, W. (2014). *AI: A Lightweight System for Tolerating Concurrency Bugs*. https://doi.org/10.1145/2635868.2635885