AKSU Journal of Engineering and Technology, 2026
Vol. 1, No. 1, Pages 34-40
Published Online April 2026 in AKSUJET (http://www.aksujet.org) ISBN: 978-978-53006-97

A HYBRID MACHINE LEARNING FRAMEWORK FOR SOFTWARE DEFECT PREDICTION USING NASA MDP DATASETS

NWACHUKWU-NWOKEAFOR, K. C.
Michael Okpara University of Agriculture, Umudike,
nwachukwu.nkenneth@mouau.edu.ng, nwachukwuken72@gmail.com.

ABSTRACT

This research presents the development of AI-based Hybrid Model for software testing and bug prediction system leveraging supervised machine learning models and advanced feature engineering techniques on the publicly available NASA Metrics Data Program (MDP) datasets. Specifically, this study presented a modular and generalizable machine learning framework that integrates multiple preprocessing techniques that include Synthetic Minority Over-sampling Technique (SMOTE), Adaptive Synthetic Sampling (ADASYN), and three distinct feature selection methods: Mutual Information (MI), Sequential Feature Selection (SFS), and the Boruta algorithm. The framework is implemented and validated across twelve NASA MDP datasets (e.g., CM1, JM1, KC1, MC1), ensuring a broad assessment of model robustness and generalizability across diverse software environments. Four classification models; Logistic Regression, Support Vector Classifier (SVC), Random Forest, and XGBoost were evaluated and compared to benchmark the prediction performance. These models were assessed using evaluation metrics that include Precision, Recall, F1-Score, Area Under the Curve (AUC), and Matthews Correlation Coefficient (MCC), with additional emphasis on addressing class imbalance and enhancing interpretability. The findings reveal that ensemble learning models, particularly XGBoost integrated with Boruta and SMOTE, consistently outperform baseline classifiers, achieving an average F1-Score of over 0.89 across most datasets while maintaining model interpretability and computational efficiency. In addition to performance benchmark, the study presents a comprehensive feature Hybrid importance analysis, identifying critical software metrics across models that contribute to early defect prediction. The research demonstrates not only the feasibility of implementing an automated. AI-based Hybrid Model for bug Testing and Prediction System in real-world settings but also contributes to a deeper understanding of software metrics and their role in predictive maintenance.

Keywords; Hybrid Model, Artificial Intelligence, Software Bug Testing and Prediction.

1. BACKGROUND

A crucial part of the software development lifecycle, software quality assurance (SQA) seeks to guarantee that software products are error-free and meet predetermined standards. As software systems get more complex, automated software defect prediction (SDP) techniques are being investigated because traditional testing methods are frequently insufficient to identify all possible problems (Challagulla *et al.*, 2008; Malhotra, 2015). Modern software systems depend heavily on software quality because errors and flaws can result in serious financial losses, harm to one's reputation, and security risks. In the United States, software failures cost businesses more than $2 trillion a year, according to research by the Consortium for information technology (IT) software quality. Most faults are discovered after the product has been deployed (CISG, 2022). Traditional methods of bug discovery and testing are not working well in light of the quick evolution of software development paradigms like Agile and DevOps. These methods, which can be laborious and prone to human mistake, frequently include post-deployment testing, static analysis, and manual code reviews. Software defect prediction is the process of identifying faulty code components by applying statistical methods and machine learning (ML). The goal of these strategies is to optimize testing efforts by predicting which modules are likely to have defects by analyzing historical data, such as source code metrics, modification history, and defect logs (Hall *et al.*, 2012; Kamei *et al.*, 2013). Over the years, various models, including traditional machine learning algorithms (e.g., Naive Bayes, Decision Trees, SVM) and, more recently, deep learning models, have been proposed for SDP (Wang and Liu, 2021). Due to their interpretability and very low computing cost, traditional machine learning models have been widely used in defect prediction. By using software measures like; cyclomatic complexity, code churn, and lines of code, techniques like Logistic Regression, Random Forest, and Naive Bayes have shown a moderate level of success in finding problematic modules (Ghotra *et al.*, 2015). However, these models might not reflect the complex interactions within the code

AKSU Journal of Engineering and Technology, 2026
Vol. 1, No. 1, Pages 34-40
Published Online April 2026 in AKSUJET (http://www.aksujet.org) ISBN: 978-978-53006-97

structure and frequently necessitate considerable feature engineering (Rahman and Devanbu, 2018). Since they can automatically acquire hierarchical features from raw data, deep learning models in particular, Convolutional Neural Networks (CNNs) and Long Short-Term Memory (LSTM) networks have been more and more popular for SDP in recent years (Zhang *et al.*, 2020). By using pre-trained language models to comprehend code semantics, transformer models such as BERT and CodeBERT have further transformed defect prediction (Feng *et al.*, 2020). By identifying both local and global dependencies in source code, transformers have demonstrated great promise in defect prediction, increasing the precision of defect detection. Although SDP has advanced, there are still a number of obstacles to overcome. While deep learning models necessitate significant processing resources and big labelled datasets, traditional models frequently suffer from the "curse of dimensionality" because of their high feature space (Arora and Sinha, 2019). Furthermore, developers may find it challenging to trust the predictions in the absence of explicit explanations because to the interpretability issues with deep learning models (Ghotra *et al.*, 2015).

- **RESEARCH GAP**

The purpose of this study is to explore how machine learning (ML) models may be used to proactively find sections of big codebases that are prone to bugs prior to code deployment. The work intends to create a prediction model that may foresee software flaws early in the development process by utilizing previous data on code changes, bug reports, and software metrics. The main objective is to raise the general caliber of software products while lowering the time and expense involved in bug fixes.

2. MATERIALS AND METHODS

This section presents a comprehensive description of the materials utilized throughout the study. These include the datasets, computational tools, programming libraries, development platforms, and the feature selection algorithms employed. Each component played a crucial role in implementing and evaluating the proposed machine learning models for bug prediction.

- **Dataset Description**

The dataset used in this study is obtained from the NASA Metrics Data Program (MDP) repository, a well-known and publicly available source of labelled software defect datasets. The datasets contain historical software metrics derived from real-world NASA software projects, with each instance representing a software module described by various static code attributes. The target variable is binary in nature, denoting whether a software module is fault-prone or non-fault-prone. The datasets are characterized by:

i. A diverse range of metrics including size, complexity (e.g., McCabe metrics), and Halstead metrics. (ii) Imbalanced class distribution, which is common in defect prediction datasets due to the naturally lower occurrence of defective modules compared to non-defective ones.

- **Tools and Libraries**

The study was implemented using Python programming language (version 3.13) due to its widespread adoption in data science and machine learning applications. A combination of scientific and machine learning libraries was used to perform pre-processing, modelling, and evaluation. These libraries include:

i. Pandas and Numpy: For data manipulation and numerical operations.
ii. Scikit-learn: For model building, evaluation, and implementation of feature selection techniques. (iii) XGBoost: For the implementation of gradient boosting models.
iii. BorutaPy: A wrapper for implementing the Boruta feature selection algorithm.
iv. Matplotlib and Seaborn: For visualization of comparative model performances and analysis
v. Imbalanced-learn: To address class imbalance via techniques such as SMOTE (if applied).

- **Computational Environment**

All experiments and model evaluations were executed in a local computing environment with the configurations:

i. Operating System: Windows 10 (64-bit)
ii. Processor: Intel® Core™ i5 @ 3.2GHz. (iii) RAM: 16GB DDR4
iii. Storage: 512GB HDD. (iv) Python Environment: Pycharm

AKSU Journal of Engineering and Technology, 2026
Vol. 1, No. 1, Pages 34-40
Published Online April 2026 in AKSUJET (http://www.aksujet.org) ISBN: 978-978-53006-97

This environment ensured optimal performance and reproducibility of results, allowing consistent evaluation across multiple experimental setups.

➢ System Model Developed.

The system model for bug prediction is a multi-phase pipeline that integrates pre-processing, feature selection, machine learning model training, and performance evaluation. The design leverages state-of-the-art algorithms to ensure data quality, dimensionality reduction, and accurate classification of software modules as fault-prone or non-fault-prone. The system model is modular, allowing for systematic experimentation and optimization at each stage. Figure1, illustrates the overall architectural flow of the system.

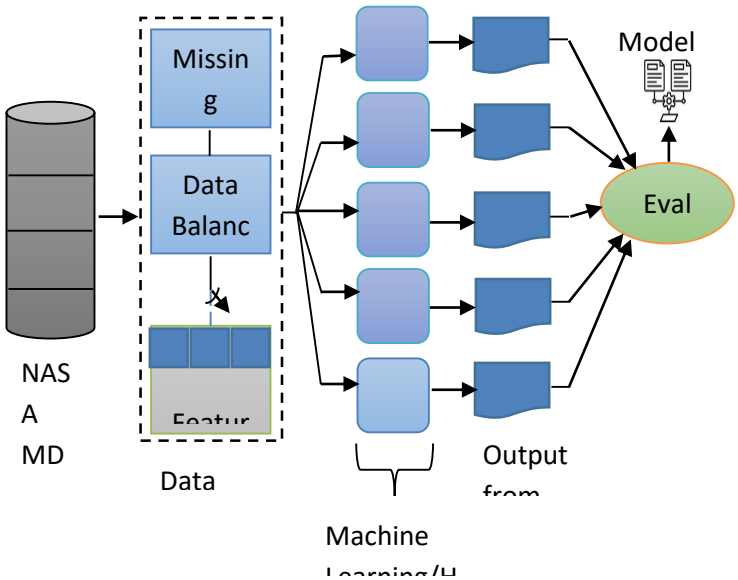


Figure 1: The proposed system model. Source: Researcher (2025)

➢ Overview of the System Model

The proposed system model comprises five major components, namely:

i. Data acquisition and pre-processing. (ii)Feature selection. (iii) Model training and classification. (iv)Model evaluation. (v) Model selection and interpretation

➢ Data Acquisition and Pre-processing

The system begins by acquiring labelled software metrics data from the NASA MDP repository, comprising both numerical and categorical features describing software modules. To ensure data quality and suitability for modelling, the following pre-processing techniques were systematically applied:

i. **K-Nearest Neighbors (KNN) Imputation:** Used to handle missing values in the dataset by estimating them based on the nearest neighbors. This technique preserves the data distribution more effectively than mean or median imputation.

ii. **Recursive Feature Elimination (RFE):** Employed as a preparatory feature reduction mechanism by recursively removing less important features based on model weights, helping to reduce over fitting and improve generalization.

iii. **SMOTE (Synthetic Minority Oversampling Technique):** Used to address class imbalance in the dataset by synthetically generating new instances of the minority class. This improves the classifier's ability to detect minority (fault-prone) modules without being biased toward the majority class at the end of this phase, a clean, balanced, and dimensionally reduced dataset is obtained, ready for feature selection and model training.

➢ Feature Selection

Following pre-processing, three prominent feature selection techniques are applied to identify the most informative attributes for bug prediction:

i. **Mutual Information (MI):** A filter-based method that quantifies the mutual dependency between each feature and the target variable. Features with high MI scores are retained for modelling.

ii. **Sequential Feature Selection (SFS):** A wrapper-based forward selection technique that incrementally builds an optimal feature subset based on model performance during cross-validation.

iii. **Boruta Algorithm:** A wrapper-based, all-relevant feature selection approach built on Random Forest, which compares actual features with randomized shadow features and retains only the statistically significant ones. These methods help reduce redundancy and irrelevance in the data, enhancing model interpretability and efficiency.

AKSU Journal of Engineering and Technology, 2026
Vol. 1, No. 1, Pages 34-40
Published Online April 2026 in AKSUJET (http://www.aksujet.org) ISBN: 978-978-53006-97

- **Model Training and Classification**

  The processed and feature-selected datasets are subsequently used to train four supervised learning models:

  (i) **Logistic Regression (LR):** A probabilistic linear classifier useful for binary classification tasks.

  (ii) **Random Forest (RF):** An ensemble method using multiple decision trees with bagging to improve robustness and accuracy.

  (iii) **Extreme Gradient Boosting (XGBoost):** An advanced boosting algorithm known for its speed and performance in structured data.

  (iv) **Support Vector Classifier (SVC):** A kernel-based classifier that constructs optimal hyper planes for classification, particularly effective in high-dimensional spaces.

  (v) **Hybrid Model:** The best performing models are selected to form the hybrid model to complement each other and enhance the general model output. Each model is trained separately on the datasets obtained from the three feature selection methods, as well as on the original dataset without feature selection for baseline comparison.

- **Model Evaluation and Selection**

After training, the performance of each model is rigorously evaluated using the following metrics:

i. **Accuracy:** Measures the overall correctness of predictions. (ii) **Precision:** Evaluates the ability of the model to identify true positives among predicted positives. (iii) **Recall:** Assesses the model's capability to detect actual positives. (iv)**F1-Score:** Harmonic mean of precision and recall, providing a balanced measure. (v)**AUC (Area-Under the Curve):** Indicating the model's discriminative ability across different threshold settings. (vi)**Training Time:** Captures the computational efficiency of each modelling approach with respect to time.

- **Implementation of Synthetic Minority Over-sampling Technique (SMOTE)**

In supervised learning, especially in real-world software engineering datasets such as the NASA MDP dataset, data imbalance is a prevalent challenge. Class imbalance, where the majority of software modules are non-defective and only a minority are defective (bug-prone), causes bias in classification models toward the majority class. To improve model performance, the data must be pre-processed, and balanced using the Synthetic Minority Over-Sampling Technique (SMOTE) and Adaptive Synthetic Sampling (ADASYN) method respectively, and explored through various visual and statistical techniques. This process ensures better insight into feature distributions, relationships, and outliers, aiding both model interpretability and accuracy. Also, this technique is deployed to handle class imbalance (bugs vs. non-bugs) and explore key patterns and relationships in the dataset. After data balancing is applied, each feature is explored using histograms or density plots to understand its distribution. For example, Lines of Code (LOC) in modules tend to have a long tail, and visualizing this helps determine the need for normalization. Let $X = \{x_1, x_2, \dots x_n\}$ be a feature such as 'LOC', and $f(x)$ be its probability density function. A kernel density estimate (KDE) is applied in this work as:

$$\hat{f}_h(x) = \frac{1}{nh}\sum_{i=1}^{n} K\left(\frac{x-x_i}{h}\right) \quad \text{Equation.1}$$

Where, K is the kernel function, h is the bandwidth, and n is the number of samples.

3. RESULTS AND DISCUSSION

In order to evaluate the performance of the proposed machine learning classification models on an both balanced and imbalanced classification problem using various pre-processing techniques, with a focus on accuracy, precision, recall, F1 score, AUC, and training time, each model was developed in python 3.13. Pycharm was used as the development environment. To aid the model development, relevant libraries were installed which include: Scikit-learn for Models, metrics, KNN imputer, SMOTE; xgboost for XGBoost classifier; imbalanced-learn for SMOTE implementation; Numpy, Pandas for Data handling; time for Timing model training; Matplotlib, Seaborn for plotting and visualization.

AKSU Journal of Engineering and Technology, 2026
Vol. 1, No. 1, Pages 34-40
Published Online April 2026 in AKSUJET (http://www.aksujet.org) ISBN: 978-978-53006-97

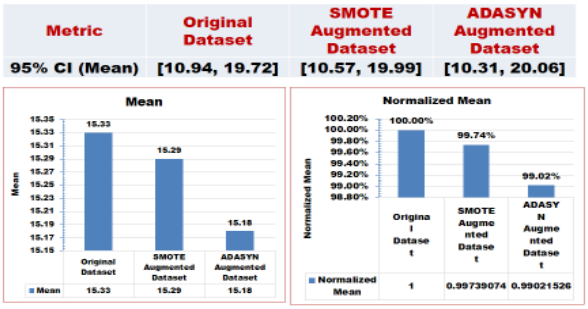


Figure 1b. Data balancing Models Performance for CM1. Files.

➢ DISCUSIONS.

Summary and Benchmark of the Proposed Hybrid Model with Standalone Models in the Work

The comparative analysis of model accuracies under different data pre-processing strategies reveals several critical insights as shown in Table 2. And Figure 1, at the baseline stage, the Support Vector Classifier (SVC) and the Hybrid Model achieved the highest accuracies (91.45%), followed closely by Random Forest (90.13%) and XGBoost (88.82%). Logistic Regression lagged considerably behind, achieving only 73.68%, reinforcing its limitation in handling imbalanced datasets without prior balancing. After applying SMOTE, accuracy improvements were observed across all models.

Table; 1. Hybrid model performance comparison across different pre-processing techniques

Evaluation Step	Accuracy	Precision	Recall	F1 Score	AUC	Training Time (s)
Baseline	0.914474	0.428571	0.250000	0.315789	0.864286	0.021351
SMOTE	0.942857	0.918919	0.971429	0.944444	0.968367	0.031486
ADASYN	0.939502	0.918919	0.964539	0.941176	0.965451	0.032176

Source: Researcher (2025)

Table; 2. Accuracy Comparison across models

Evaluation Step	Logistic Regression	Random Forest	XGBoost	SVC	Hybrid Model
Baseline	0.736842	0.901316	0.888158	0.914474	0.914474
SMOTE	0.875000	0.928571	0.939286	0.878571	0.942857
ADASYN	0.850534	0.925267	0.935943	0.864769	0.939502

Source: Researcher (2025).

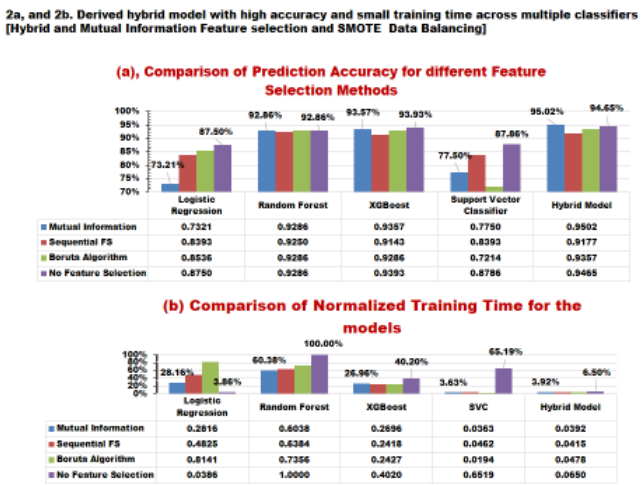
Table; 3. Training Time Comparison

Feature Selection Method	Logistic Regression	Random Forest	XGBoost	Support Vector Classifier

AKSU Journal of Engineering and Technology, 2026
Vol. 1, No. 1, Pages 34-40
Published Online April 2026 in AKSUJET (http://www.aksujet.org) ISBN: 978-978-53006-97

Mutual Information	0.198375	0.425387	0.189900	0.025558
Sequential FS	0.339895	0.449767	0.170356	0.032553
Boruta Algorithm	0.573516	0.518239	0.170969	**0.013676**
No Feature Selection	0.027222	0.704490	0.283173	0.459224

Source: Researcher (2025)



Figure; 2a and 2b. Derived Hybrid Model with high accuracy and small training time across multiple classifiers (Hybrid and Mutual Information Feature Selection and SMOTE Data Balancing).

4. CONCLUSION AND CONTRIBUTIONS.

The Hybrid Model, particularly when combined with SMOTE, provided a balanced trade-off across all metrics, underscoring the utility of ensemble learning in complex classification tasks. From a computational perspective with respect to the Tables, 1, 2, and 3 above, the following conclusions and contributions can be adduced;

(i) Logistic Regression and SVC were fastest to train but at the cost of lower initial performance.

(ii) XGBoost and Random Forest had longer training times, especially after applying SMOTE or Boruta, due to model complexity and data augmentation.

(iii)Feature selection methods, notably Boruta, introduced additional training time overhead (e.g., Logistic Regression increased from 0.027s (no FS) to 0.573s (Boruta)), but this was justifiable given the performance gain.

➢ LIMITATIONS OF THE WORK.

A balance between model complexity, training cost, and predictive gain must be contextually evaluated, particularly in resource-constrained environments, these factors constitutes the major limitations of the work.

➢ MAJOR CONTRIBUTION

Hybrid modeling, by integrating strong learners like XGBoost with linear classifiers such as Logistic Regression, offers a powerful general-purpose solution with minimal compromised performance. The modeling framework developed and tested in this study offers a highly effective and generalizable pipeline for software defect prediction in imbalanced environments. From the graphical figures 1b, 2a and 2b above, the best performance was obtained using a hybrid model (XGBoost + Logistic Regression) with SMOTE balancing, achieving 94.28% accuracy, 91.89% precision, 97.14% recall, and 94.44% F1-score. These results provide a strong foundation for deploying robust defect prediction systems in real-time as shown in Table 3. This is an achievement in real-world software engineering workflows.

➢ SUGGESTED FUTURE WORK

The use of Deep Learning and Real-time System for an optimized result and improvement to this work is suggested, especially

considering the current research trajectories in Neural Networks and Genetic Algorithms.

REFERENCES

Charles, Q., and Perl, Y. (2024). A Deep Learning Approach for Software Defect Forecasting Using Adaptive Neural Networks. IEEE, 8, 583-695.

Capgemini, and Sogeti. (CISG, 2022). World Quality Report 2022-23. Retrieved from www.capgemini.com

Choudhary, G., Goraya, M. S., and Sikka, G. (2020). Software test case prioritization: A systematic mapping study. Journal of Systems and Software, 161, 110463.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., and Zhou, M. (2020). CodeBERT: A pre-trained model for programming and natural languages. Findings of the Association for Computational Linguistics: EMNLP 2020, 1536–1547.

Zhang, Y., Wang, Q., and Harman, M. (2020). Explainable software analytics. IEEE Software, 37(4), 46–54. https://doi.org/10.1109/MS.2020.2988351

Arora, A., and Sinha, D. (2019). Software testing techniques for test case generation using machine learning. International Journal of Innovative Technology and Exploring Engineering (IJITEE), 8(12), 4152–4156.

Rahman, F., and Devanbu, P. (2018). Just-in-time defect prediction: Using deep learning and data mining to identify bug-prone files. Journal of Software: Evolution and Process, 30(4), 18-24.

Malhotra, R. (2015). A systematic review of machine learning techniques for software fault prediction. Applied Soft Computing, 27, 504–518.

Ghotra, B., McIntosh, S., and Hassan, A. E. (2015). Revisiting the performance of defect prediction models. Proceedings of the 37th International Conference on Software Engineering (ICSE), 602–613.

Ghotra, B., McIntosh, S., and Kamei, Y. (2015). A Large-Scale Study of the Impact of Feature Selection Techniques on Defect Classification Models. Proceedings of the IEEE/ACM International Conference on Software Engineering, 146-156.

Kamei, Y., Shihab, E., Adams, B., Hassan, A. E., Mockus, A., Sinha, A., and Ubayashi, N. (2013). A Large-Scale Empirical Study of Just-in-Time Quality Assurance. IEEE Transactions on Software Engineering, 39(6), 757-773.

Hall, T., Beecham, S., Bowes, D., Gray, D., and Counsell, S. (2012). A Systematic Literature Review on Fault Prediction Performance in Software Engineering. IEEE Transactions on Software Engineering, 38(6), 1276-1304.