Available online at sjuoz.uoz.edu.krd

journals.uoz.edu.krd

Science Journal of University of Zakho

Vol. 13, No.4 pp449–463October-December,2025

p-ISSN: 2663-628X
e-ISSN: 2663-6298

HYBRID TECHNIQUE FOR SOFTWARE DEFECT PREDICTION USING MACHINE LEARNING TECHNIQUES

Darius T Chinyio [1, *], Martin E Irhebhude [1], and Muhammad Jumare Haruna[2]

[1] Department of Computer Science, Nigerian Defence Academy, Kaduna, Nigeria.

[2] Department of Computer Science, Federal University of Education, Zaria, Nigeria.

*Corresponding author email: dtchinyio@nda.edu.ng

Received: 29 Mar 2025 Accepted:15 Jun 2025 Published:1Oct 2025 https://doi.org/10.25271/sjuoz.2025.13.4.1532

ABSTRACT:

Human errors during software development lead to many defects, which emphasizes the importance of early detection and minimization. However, existing approaches often fall short in delivering accurate, scalable, and generalizable predictions due to challenges such as class imbalance, feature extraction limitations, and computational inefficiencies. This study proposes a hybrid method using a Convolutional Neural Network (CNNs) + Long Short-Term Memory (LSTM) for feature extraction, addressing class imbalance with Adaptive Synthetic Sampling (ADASYN) and subsequent training using Extreme Gradient Boosting (XGboost), to predict software defects. The proposed approach was evaluated on five publicly available datasets (CM1, MC1, KC1, PC1, and PC4) and compared with state-of-the-art (SOTA) models. Experimental results demonstrated that the hybrid model significantly outperforms traditional XGBoost-based models in terms of recall, F1-score, and area under the receiver operating characteristic curve (AUC), addressing the shortcomings of existing methods. Results demonstrate the effectiveness of the proposed method, with notable performance metrics achieved across all datasets. For example, on the MC1 dataset, the model attained an accuracy of 0.9980, a precision of 0.9971, a recall of 0.9988, an F1-score of 0.9980, and an AUC-ROC of 0.9999. On the KC1 dataset, it achieved an accuracy of 0.9344, a precision of 0.9265, a recall of 0.9375, an F1-score of 0.9320, and an AUC-ROC of 0.9839. The model achieves better performance than traditional machine learning methods and separate deep learning models, especially in the areas of recall and AUC-ROC. This research presents a robust solution through hybrid approaches that address class imbalance and maintain high predictive accuracy for software development process tasks, offering insights into the trade-offs between machine learning and deep learning methods.

KEYWORDS: Software Defect Prediction (SDP), CNN, LSTM, Machine learning, Deep Learning, Hybrid Technique, XGboost

1. INTRODUCTION

Software defects represent faults in computer programs that may lead to system failures, data loss, security vulnerabilities, and financial losses (Elentukh, 2023; Shafiq *et al.,* 2023). While the terms 'defect', 'bug', and 'error' are sometimes used interchangeably, a defect generally refers to an imperfection in code functionality that may or may not result in a bug, which is an observable deviation from expected behavior during execution. An error, on the other hand, typically refers to a human mistake made during development that leads to defects in the code.

Defects not only affect runtime performance and reliability but also compromise key software design principles such as modularity and separation of concerns. Faulty modules are less likely to be reused due to their instability or unclear functionality, which reduces maintainability and increases technical debt. In this study, the static code features extracted from the datasets (e.g., cyclomatic complexity, coupling, cohesion) reflect structural weaknesses that are closely tied to defect proneness and negatively influence reusability and modular design.

According to research by Krasner (2021), Mahmoud *et al.* (2024), and Mehmood *et al.* (2023), software defects account for half of project expenses, while also causing system breakdowns and security risks, with additional negative impacts on user satisfaction. Despite the advancement of machine learning, the current approaches to software defect prediction still struggle with issues such as the existence of imbalanced datasets (Saidani *et al.,* 2022; Giray *et al,* 2023), difficulties in comprehending intricate code forms (Daneshdoost & Feyzi, 2023; Wan *et al.,* 2024), poor project generalization (Nevendra & Singh, 2022), and insufficient explanation of the prediction (AL-Hadidi & Hasoon, 2024). The comprehension of software defects from a

[*] Corresponding author

theoretical perspective has changed from mere bug tracking to more advanced predictive analytics, where scholars have sought to develop defect explanation frameworks and models that capture a number of attributes (Vogel-Heuser *et al.,* 2015). There has been a shift from conventional statistical methods to more advanced ones based on machine learning (ML) in software defect prediction (SDP), with the use of algorithms and deep learning being more effective in dealing with complex code structures. (Kumar *et al.,* 2023; Pachouly *et al.,* 2022). Understanding the relationship between software complexity and the likelihood of defects occurring is a prominent theoretical perspective. Numerous measures have been proposed to quantify the complexity of software systems and their correlation with error occurrence, including object-oriented design metrics, Halstead's software science metrics, and McCabe's cyclomatic complexity (Kumar *et al.,* 2023). Furthermore, external process-related elements, including a change's history, a developer's experience, and the methodologies employed in the development, have been deemed important in forecasting software defects (Pachouly *et al.,* 2022). There remain several research gaps and emerging trends; however, the advancement of software defect prediction methodologies has not kept pace with (Olaleye *et al.,* 2023). Although ML and deep learning (DL) models have improved prediction accuracy, their practical application is still hindered by the models' inherent complexity and the resulting lack of understanding about them (Patil *et al*., 2024). Many current models show robust performance on particular datasets, but these models often fail to generalize across different projects and domains (Alzeyani & Szabó, 2024). Meeting the temporal dimensions of software development poses another critical challenge owing to the changes in software characteristics and defect patterns over time (Kaliraj & Thomas, 2024). The latest trend reported by Khan and Masum (2024) is the incorporation of software defect prediction tools within the continuous integration and deployment pipelines for the purpose of real-time feedback and proactive defect mitigation. This new direction of research in software defect prediction prepares the ground for further efforts that bound these limitations of accuracy, generalizability, and practicability of predictive software system defects for automation. Likewise, the integration of deep learning technologies into algorithms created for predicting software bugs has opened wider opportunities. Convolutional CNNs and LSTMs are outstanding examples of modern applicative deep learning models that have achieved significant breakthroughs in time series prediction, image recognition, and natural language processing (Goodfellow *et al.,* 2016). Because these models automatically capture the structure of data as a hierarchy of features, they are particularly powerful for quite complex tasks where input variables interact with each other in many ways. Even with their unparalleled advantages, deep learning models often require substantial computational power and large amounts of training data, which are not always readily available in software defect prediction scenarios (Wang *et al.,* 2022). To achieve the objectives, experimental evaluations were conducted using multiple datasets commonly referenced in the field, including CM1 and MC1, as well as other datasets such as KC1, PC4, and PC1. The models' performances were assessed using performance metrics. A significant contribution of this study is the development of a hybrid framework that combines the strengths of deep learning for capturing complex spatial and temporal dependencies in software metrics with the efficiency and robustness of XGboost for classification, particularly in imbalanced datasets. The proposed method demonstrates superior performance compared to traditional ML models and standalone DL architectures, particularly in terms of recall and AUC-ROC scores. These metrics are especially important in defect prediction, where identifying all defective modules (high recall) and achieving strong class separation (high AUC) are critical for practical deployment. The rest of the paper is structured: Related works are covered in Section 2. In contrast, section 3 materials and methods offer a thorough explanation of the approach, covering feature extraction, dataset preparation, data balancing, model assessment, and experimental setup. The experimental results are shown in Section 4, along with a discussion of the findings in relation to previous research and state-of-the-art models. The work is finally concluded, and future research concerns are outlined in Section 5.

2. REVIEW OF RELATED WORK

Software engineering research has placed much emphasis on predicting software problems, with many papers examining different methods to increase forecast accuracy and dependability. This section looks at earlier research that has significantly advanced the field, with a focus on traditional ML methodology, deep learning techniques, and hybrid approaches. The study by (Ali *et al.,* 2024) highlighted the importance of software defect prediction (SDP) in enhancing software quality and reducing testing costs by identifying and prioritizing defective modules for testing. Preprocessing (splitting, cleaning, and normalization), classification using four different supervised machine learning classifiers, ensemble modelling using a voting ensemble strategy, and data collection from seven historical defect datasets are all included in the methodology. The results demonstrated that the VESDP model outperformed twenty cutting-edge defect prediction methods with impressive accuracy of 86.87%, 79.12%, 68.42%, 89.33%, 92.16%, 87.97%, and 87.14% across the CM1, JM1, MC2, MW1, PC1, PC3, and PC4 datasets, respectively. The study also lacked detailed results on the computational complexity or scalability of the VESDP model, which are critical for large-scale projects, and does not extensively discuss the impact of data quality, including noise or missing values, on model performance. A modified Random Forest-based technique for software fault prediction was implemented using the JM1 dataset, which comprises various software metrics indicating the presence or no presence of a defect in a module (Kaliraj & Thomas, 2024). The study examined issues such as feature selection, imbalanced datasets, model overfitting, data scarcity, and interpretability challenges in software error prediction. Random Forest was used for feature selection, SMOTE was used to solve class imbalance, preprocessing was used to manage null values and data problems, and the Random Forest classifier was used to handle high-dimensional datasets and reduce overfitting. However, the approach's generalizability to other datasets (AL-Hadidi & Hasoon, 2024) Alternatively, contexts are limited by their concentration on a single dataset, and they ignore potential drawbacks such as computational cost, memory needs, or the demand for intensive hyperparameter adjustment.

A study by AL-Hadidi & Hasoon, (2024) Software defect prediction was implemented using XGboost with hyperparameter optimisation for their experimental analysis. The research study sought to enhance prediction accuracy and performance by applying ensemble learning methods together with optimization techniques. The research leverages advanced techniques such as XGboost and grid search with cross-validation for hyperparameter tuning while also addressing dataset imbalance using oversampling. The researchers employed several NASA MDP datasets to expand the applicability of their findings. The optimization process resulted in a notable performance increase for the model, with accuracy improving from 0.888 to 0.938 for the CM1 dataset and from 0.743 to 0.795 for the KC1 dataset. The study suffers from limitations, which include insufficient exploration of various data balancing strategies, along with an exclusive focus on XGboost without testing other advanced methods, and a lack of detailed analysis on the computational costs of hyperparameter adjustments.

Alkaberi & Assiri, (2024) focused on utilizing CNN and multilayer perceptron (MLP) to detect software errors in order to improve software quality. The data was pre-processed using SMOTEND oversampling, log transformation, and standardization, and they employed 12 datasets with 20 object-oriented metrics from the PROMISE repository. Kendall's correlation coefficient and mean squared error (MSE) were the evaluation metrics utilized. Before applying SMOTEND, CNN achieved MSE=1.316 and Kendall=0.162 on test data, while MLP achieved MSE=1.73 and Kendall=0.183. After addressing data imbalance with SMOTEND, performance improved significantly CNN achieved MSE=0.218 and Kendall=0.363, while MLP performed better with MSE=0.195 and Kendall=0.416 on test data. These were compared to baseline machine learning models: decision tree regression (DTR) achieved MSE=0.17 and Kendall=0.486, while support vector regression (SVR) showed poorer performance with MSE=0.257 and Kendall=0.276 on balanced test data. The study validated the approach, but limitations include potential external validity constraints and a lack of ablation studies. Additional metrics, such as model complexity and inference time, could enhance the evaluation. Ponnala & Reddy, (2023) using method-level features from an open-source Java e-commerce project, developed an ensemble model for software defect prediction. The study combined random forest, SVM, and LightGBM algorithms using logistic stacking to improve prediction accuracy. Key strengths include the use of fine-grained method-level metrics (75 features reduced to 25 via PCA) and the ensemble approach, which outperformed individual models with an ROC AUC of 0.853 and 81% accuracy. However, limitations include analysis of only one project, a lack of comparison with state-of-the-art ensemble techniques, and insufficient discussion of practical implications. The study demonstrated the potential of ensemble methods and method-level features for defect prediction, but further validation across diverse projects and exploration of feature importance would enhance its impact. Future work could focus on model interpretability and integration into development processes. Maddipati & Srinivas, (2021) reported a way to improve software fault prediction by tackling the issues of excessive dimensionality and class imbalance. The research employed dimensionality reduction through a statistical technique, that is to say, utilizing a method to simplify complex data structures, combining this with an ensemble approach paired with an adaptive fuzzy system. When compared to current approaches, the methodology increased the AUC by 15%, indicating greater predictive accuracy. While the results were promising, demonstrating the model's effectiveness in NASA datasets, the study's reliance on specific datasets limits its generalizability. Additionally, it did not explore the impact of varying software projects or defect densities on performance. The study offered a significant advancement in balancing accuracy and cost-effectiveness in SDP, though further research is needed for broader applicability. Olorunshola *et al*., (2020) evaluated various machine learning classification algorithms to identify the best performer in predicting software defects, emphasizing the importance of minimizing misclassification to avoid wasted developer effort. Using WEKA version 3.8.3 and the JM1 dataset, the study assessed twelve algorithms from six categories. Standard Performance metrics included accuracy, false positive rate, Kappa statistic, RMSE, among others, with a primary assessment through 10-fold cross-validation. The study found that the Random Forest algorithm outperformed most others, while the Bayes Net classifier excelled in terms of the false positive rate, achieving the lowest FP-rate of 0.391. A comprehensive evaluation of various classification algorithms, including less commonly explored ones, is a notable strength. However, the study's drawback includes its focus on a single dataset, raising concerns about the generalizability of the results. The rationale for selecting specific algorithms and metrics was not clearly explained, and the study lacked detailed analysis beyond presenting numerical performance metrics. Recent literature in software defect prediction highlights several critical gaps in current methodologies. While studies have advanced predictive modelling capabilities, they often employ single-algorithm solutions that fail to address the multifaceted challenges in defect prediction comprehensively (AL-Hadidi & Hasoon, 2024; Wang *et al*., 2022). Key limitations include inadequate handling of class imbalance, computational inefficiency, and poor project generalizability (Jin, 2021; Khalid *et al*., 2023). The current state of software defect prediction still clearly lacks a fully comprehensive approach that might actually bring together many ensemble techniques, various optimization methods, and some data balancing procedures. Much of the existing research has just focused on individual components rather than addressing these elements in a more unified way, which has eventually led to many disconnected solutions that currently fail to fully maximize both predictive capabilities and real-world implementation (Shen & Chen, 2020; Tameswar *et al*., 2022). Additionally, to effectively balance accuracy, computational efficiency, and model generalization in SDP applications, the field clearly needs to develop a more holistic methodology

3. MATERIALS AND METHODS

The current section explains how the study model was build using a hybrid-type approach that essentially combines both deep learning feature extraction with the XGboost algorithm to help with Software Defect Prediction. This particular approach uses a hybrid CNN-LSTM model to fully analyze many numerical software metrics Dataset. The CNNs identify some local patterns, while the LSTMs still process much of the sequential data relationships. Additionally, the extracted features are eventually fed into XGboost for training to help with defect classification.

Dataset Description and Preprocessing:

The study utilized five of the most established benchmark datasets (which include CM1, KC1, PC1, PC4, and MC1) that contain many static code measurements like cyclomatic complexity, some Halstead metrics, and various code line counts. These particular datasets, which are still widely used in much of the defect prediction research, include both the metric attributes and also the defect labels for each of the software modules (AL-Hadidi & Hasoon, 2024; Ali *et al*., 2024; Menzies *et al*., 2015; Shepperd *et al*., 2018).

Table 1: Dataset Description

Dataset	Type	No of features	Programming Language	Instances	Non-Defective	Defective
CM1	Procedural	22	C	327	285	42
MC1	Object Oriented	39	C++	8737	8669	68
PC1	Procedural	22	C	735	674	61
PC4	Procedural	37	C	1379	1201	178
KC1	Object Oriented	22	C++	2095	1770	325

The SDP model uses various data preprocessing steps to fully enhance both data quality and model effectiveness. Before feeding the features into the CNN-LSTM model, the following preprocessing steps were applied:

The process begins by importing the dataset and converting any missing values marked with "?" into the NaN format for much better processing. To handle these missing values, the model uses mean substitution, which helps to maintain more data consistency (Ghotra *et al*., 2017). Additionally, all features are converted to numeric format, and the target variable undergoes binary encoding (0 and 1) using LabelEncoder, with defective 1 and non-defective 0, which aligns with established classification methods (Farabet *et al*., 2013). Further, feature scaling is implemented through StandardScaler, which basically normalizes the numerical features to a mean of 0 and standard deviation of 1, because this prevents any single feature from becoming too dominant in the model, hence improving model convergence (Ahmed *et al*., 2023). Due to the dataset imbalance between defective and non-defective instances, the Adaptive Synthetic Sampling technique creates synthetic minority class samples to achieve a much better balance. This resampling approach helps to enhance both the model's learning capabilities and detection accuracy (Jude & Uddin, 2024). Moreover, the process includes verification of balanced class distribution after resampling by comparing the defect and non-defect instances. These comprehensive preprocessing steps, which include missing value treatment, categorical encoding, feature scaling, and class balance correction, are clearly vital for achieving optimal model performance in defect prediction tasks (Goyal, 2022; Hussein *et al*., 2020; Jude & Uddin, 2024). The dataset description after applying ADAYSN is reported in Table 2.

Table 2: Data Description After Applying ADAYSN

Dataset	Instances	Non–Defective	Defective
CM1	561	285	276
MC1	17340	8669	8671
PC1	1345	674	671
PC4	2443	1201	1242
KC1	3501	1770	1731

Addressing Class Imbalance:

The issue of class imbalance commonly seen in software defect datasets was handled using the ADASYN approach, which created artificial samples for underrepresented categories. This improved prediction performance overall while enhancing the capability to learn from examples that are not well-represented (Hussein *et al*., 2020). The ADASYN method adapts to generate synthetic data points for minority classes (Hussein *et al*., 2020). The number of artificial samples generated depends on how difficult it is to classify specific minority cases, meaning harder cases get more synthetic examples. Figure 2 demonstrates the preprocessing approach applied to the dataset. Algorithm 1 (Hussein *et al*., 2020) outlines the process: to put it simply, the method evaluates data distribution patterns first, then calculates neighborhood relationships between samples, and finally produces new instances proportionally based on the complexity of classification tasks.

Algorithm 1

Step 1: Defining Classification Complexity:

The classification challenge for underrepresented data points is measured by examining the proportion of surrounding examples that belong to dominant categories compared to all nearby instances. In equation 1, let r_i represent this ratio:

$$\mathrm{r_i} = \frac{\text{Number of majority class neighbors of } \mathrm{x_i}}{\text{Total number of neighbors of } \mathrm{x_i}} \qquad (1)$$

Where:

r_i Is the ratio, for instance x_i.

The number of neighbors is typically determined using k-nearest neighbors (k-NN).

Step 2: Compute the Total Number of Synthetic Samples to Generate:

The total number of synthetic samples that need to be created for the minority class is N_{syn} Equation 2 is used to calculate this.

$$\mathrm{N_{syn}} = \mathrm{N_{maj}} - \mathrm{N_{min}} \qquad (2)$$

Where:

N_{maj}: The majority class's number of instances.

N_{min}: The minority class's total number of instances.

Step 3: Determine the Number of Synthetic Samples for Each Minority Instance:

The quantity of synthetic samples G_i t to be produced for each minority case x_i is directly proportional to its difficulty ratio r_i as indicated in equation 3.

$$G_i = \left\lfloor N_{\text{syn}} \cdot \frac{\lfloor r_i \rfloor}{\left\lfloor \sum_{j=1}^{N_{\text{min}}} r_j \right\rfloor} \right\rfloor \qquad (3)$$

Where:

G_i: "Number of synthetic samples for instance x_i" .

N_{syn}: Total number of synthetic samples needed to balance the dataset.

r_i: Difficulty ratio of the minority instance x_i, which represents the proportion of majority class neighbors among its k-nearest neighbors.

$\sum_{j=1}^{N_{\text{min}}} r_j$: Sum of the difficulty ratios for all instancesin the minority class.

$\frac{r_i}{\sum_{j=1}^{N_{\text{min}}} r_j}$: Normalized difficulty ratio of x_i, indicating the relative difficulty of learning x_i compared to other minority instances.

Step 4: Generate Synthetic Samples:

Create G_i synthetic samples for every minority instance x_i, by interpolating with its closest neighbors. Equation 4 creates a synthetic sample.

$$\mathrm{x_{syn}} = \mathrm{x_i} + \lambda \cdot (\mathrm{x_z} - \mathrm{x_i}) \qquad (4)$$

Where:

$\mathrm{x_{syn}}$: The newly generated synthetic sample for the minority class.

$\mathrm{x_i}$: The original minority instance for which synthetic samples are being generated.

x_z: A randomly selected neighbor of $\mathrm{x_i}$ from its k-nearest neighbors in the minority class.

λ: A random number drawn from the uniform distribution is called Lambda. The synthetic sample's location along the line segment between $\mathrm{x_i}$ and $\mathrm{x_z}$ is determined by this parameter. By varying λ, the synthetic sample is interpolated to a location between $\mathrm{x_i}$ and $\mathrm{x_z}$, ensuring diversity in the generated data.

Hybrid CNN-LSTM Feature Extraction:

For feature extraction, the hybrid approach makes use of CNN and LSTM's advantages. To extract spatial information from the input data, a CNN is used. Equation 5 (Goodfellow *et al*., 2016) Illustrates the general mathematical formulation for a single convolutional layer:

$$Y = f(W * X + b) \qquad (5)$$

Where:

X is the input feature map (e.g., a matrix representing an image or software metric data).

W is the learnable convolutional kernel (or filter), which slides over the input feature map to extract local patterns. Each filter learns specific spatial features during training.

$*$ denotes the convolution operation.

b is the bias term, is appended to the convolution operation's output to improve the model's fit to the data.

f is the activation function, typically a non-linear function applied element-wise to introduce non-linearity into the model. It helps mitigate issues like vanishing gradients and improves convergence during training (Goodfellow *et al*., 2016)

LSTM is used to model temporal dependencies and sequential patterns. An LSTM (Long Short-Term Memory) cell updates its hidden state h_t which represents the output of the LSTM cell at time t, capturing the relevant information from the sequence up to that point, and the memory cell C_t which acts as the "memory" of the LSTM, storing long-term dependencies across time steps. This is shown using the following mathematical functions:

Forget Gate Activation (f_t)

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right) \qquad (6)$$

The amount of information from the preceding memory cell C_{t-1} that should be remembered or forgotten is decided by the forget gate. The bias term is b_f and while the forget gate's weight matrix W_f. The sigmoid activation function, or σ, produces values ranging from 0 to 1, signifying the extent to which each component of the cell state is forgotten

Input Gate Activation (i_t)

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \quad (7)$$

The input gate determines the amount of new data that should be added to the memory cell from the current input x_t. b_i is the bias term and W_i is the input gate's weight matrix.

Information flow is controlled by the sigmoid function, which makes sure the gate outputs values between 0 and 1.

Candidate Memory Cell ($\widetilde{C}_t$)

$$\widetilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \quad (8)$$

The candidate memory cell computes a potential update to the memory cell C_t. The bias term is b_C, while the weight matrix for the candidate memory cell W_C. The updates are bounded because the hyperbolic tangent (tanh) activation function generates values in the$[-1, 1]$ range.

Memory Cell Update (C_t)

$$C_t = f_t \odot C_{t-1} + i_t \odot \widetilde{C}_t \quad (9)$$

The new information is combined with the prior memory state C_{t-1} to update the memory cell C_t

$f_t \odot C_{t-1}$: The amount of the prior memory C_{t-1} that is kept is decided by the forget gate f_t

$i_t \odot \widetilde{C}_t$: i_t controls how much of the new candidate memory $\widetilde{C}_t$ is added to the memory cell. $\odot$ denotes element-wise multiplication, allowing fine-grained control over the memory updates.

Output Gate Activation (o_t)

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \quad (10)$$

How much of the updated memory cell C_t will be shown as the hidden state h_t is decided by the output gate. b_o is the bias term, and W_o is the output gate's weight matrix The sigmoid function ensures that the output gate outputs values between 0 and 1, controlling the exposure of the memory cell.

Hidden State Update (h_t)

$$h_t = o_t \odot \tanh(C_t) \quad (11)$$

The hidden state h_t is computed based on the updated memory cell C_t and the output gate o_t.

$\tanh(C_t)$: The hyperbolic tangent function applies a non-linear transformation to the memory cell, ensuring that the output is bounded between -1 and 1. $o_t \odot \tanh(C_t)$: The output gate o_t modulates the transformed memory cell, determining the final hidden state

Combining both of these models, as shown in Figure 1, enables a thorough representation of the input data, which can improve XGboost's performance (D. Wang *et al*., 2022; H. Wang *et al*., 2021; S. Wang *et al*., 2022). Each dataset is pre-processed to ensure consistency in feature scaling and encoding. Missing values are handled using imputation techniques, and categorical variables are encoded using one-hot encoding. Spatial features are extracted from the input data using a convolutional layer. Three (3) filter levels were applied to the data in order to capture the various degrees of abstraction. To capture sequential dependencies, an LSTM layer is applied to the CNN layer's output. This step is particularly useful for datasets with time-series or ordered features. The outputs from the CNN are passed to the LSTM layers, which serves as input for the subsequent ML models. Given that software defect datasets primarily contain numerical attributes, the CNN-LSTM model extracts statistical and temporal features (Khleel & Nehéz, 2022). The architecture implements a sophisticated sequential processing pipeline for defect prediction. The model begins by accepting a one-dimensional input vector of shape (100,) this means each input instance is a single row vector containing 100 values (features) which is then reshaped to (100,1) to accommodate the convolutional operations. A Conv1D layer with 64 filters and a kernel size of 3 performs the first feature extraction. It uses ReLU activation to identify non-linear patterns in the input data and to identify spatial correlations between software metrics. A MaxPooling1D layer with a pool size of two comes next, which minimizes the spatial dimensions without sacrificing important features. To produce a compact feature representation, a second convolutional block with the same configuration as the first one further processes the down-sampled features. This is followed by another MaxPooling1D layer. To reduce overfitting, a dropout layer (rate=0.5) is incorporated (Charles, 2024). The processed features then flow through a dual LSTM structure, where the first LSTM layer maintains temporal sequences by returning an output of shape (49, 50), while the second LSTM layer consolidates this information into a final feature vector of shape (50,). This models the sequential dependencies inherent in software defect metrics. The architecture culminates in a Dense layer with Sigmoid activation, producing a binary classification output of shape (1, 0), effectively predicting the presence or absence of defects, the features extracted are then used to for subsequent model training using machine learning algorithm XGboost

The hyperparameters were selected based on empirical experimentation and performance validation across multiple datasets. Initial values were selected following commonly used configurations in similar sequence modeling tasks (e.g., time-series classification and NLP), with further refinements made through iterative training and validation to optimize recall and AUC-ROC performance. The Conv1D layer with 64 filters and a kernel size of 3 was chosen for its ability to capture local patterns without excessive computational cost. ReLU activation was used to introduce non-linearity while avoiding vanishing gradients. A dropout rate of 0.5 was applied to reduce overfitting during training. The dual LSTM structure was selected to model sequential dependencies effectively, where the first. LSTM layer preserves temporal information (output shape: (49, 50)), and the second layer produces a compact feature vector (output shape: (50,)) for final classification and subsequent feature representation.

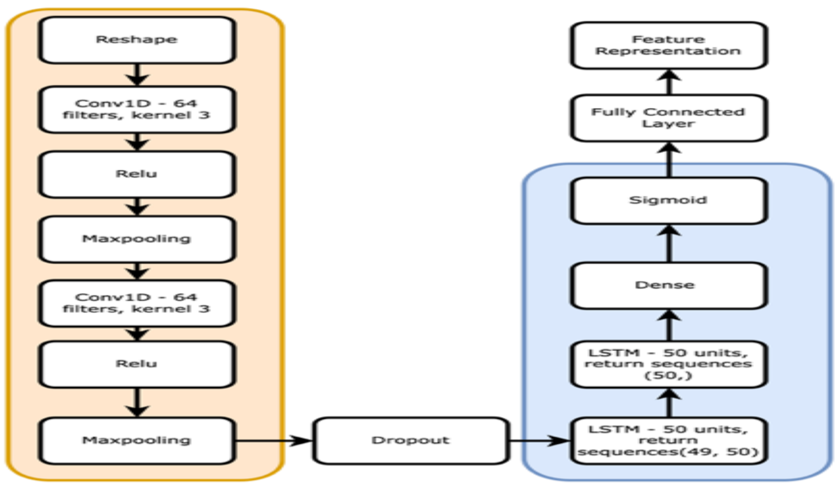

Reshape
Conv1D - 64 filters, kernel 3
Relu
Maxpooling
Conv1D - 64 filters, kernel 3
Relu
Maxpooling
Dropout
LSTM - 50 units, return sequences(49, 50)
LSTM - 50 units, return sequences (50,)
Dense
Sigmoid
Fully Connected Layer
Feature Representation

Figure 1: CNN LSTM Architecture

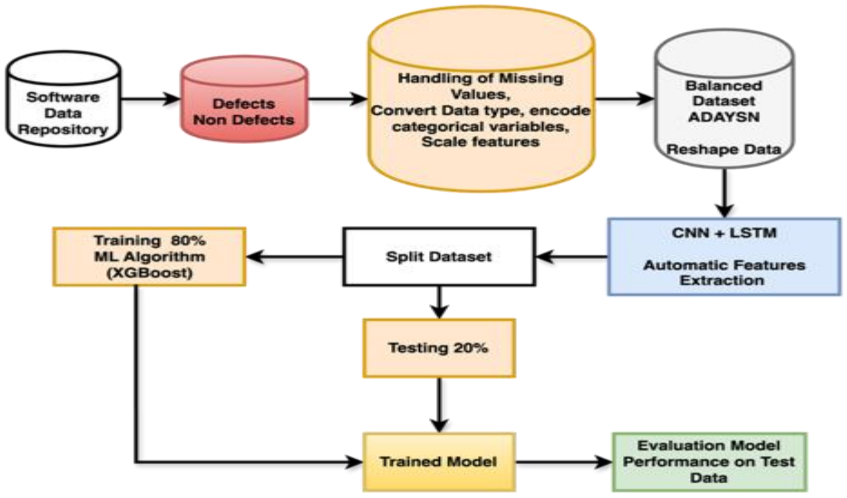

Software Data Repository
Defects Non Defects
Handling of Missing Values, Convert Data type, encode categorical variables, Scale features
Balanced Dataset ADAYSN
Reshape Data
CNN + LSTM
Automatic Features Extraction
Split Dataset
Training 80% ML Algorithm (XGBoost)
Testing 20%
Trained Model
Evaluation Model Performance on Test Data

Figure 2: Proposed Methodology

Types of Features Extracted:

Each dataset includes numerical features that quantify different characteristics of the software modules. These features can be broadly categorized into the following groups:

Cyclomatic Complexity Metrics: They measure the complexity of a program by counting the number of linearly independent paths through the source code. Higher values often correlate with an increased likelihood of defects due to the difficulty in testing and maintaining such code.

Halstead Metrics: These are based on the number of operators and operands in the code, including: Program length, Vocabulary size, Volume, Difficulty and Effort. These metrics estimate the effort required to understand or debug the code and are useful indicators of potential defects.

Lines of Code (LOC):

This feature represents the total number of lines in a module, which may indicate the size and complexity of the code. Larger modules tend to have more defects due to increased maintenance and readability challenges.

Object-Oriented Metrics (for OO datasets like KC1 and MC1): These include: Number of classes, Number of methods per class, Depth of inheritance tree, Coupling between objects, and Response for a class. These metrics help assess the design quality and potential fault-proneness of object-oriented systems.

Code Churn or Change Frequency:

Some datasets include historical data on how frequently a module has been changed or modified, which is a known indicator of instability and potential defects.

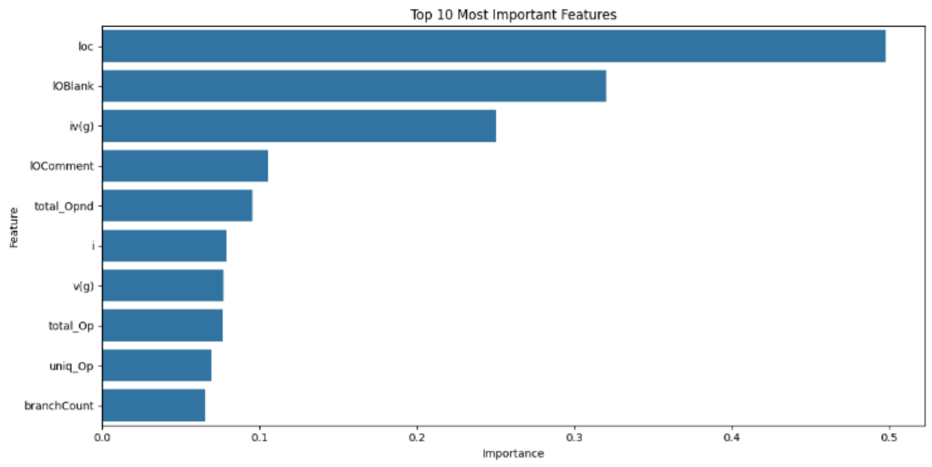


Figure 3: Feature Importance

Training and Evaluation:

The XGboost classifier machine learning model was employed in this study. In order to guarantee thorough analysis, every dataset was methodically divided into training and testing sets using an 80-20 split ratio, which enabled reliable model evaluation and validation. While the paper emphasizes the use of static code metrics such as cyclomatic complexity, Halstead metrics, lines of code, coupling, and cohesion for software defect prediction, it is important to clarify the nature and strength of the relationship between these metrics and the model's predictions. To address this, a post-hoc feature importance analysis was conducted using the trained XGBoost classifier, which provides insights into how each metric contributes to the final classification decision, as shown in Figure 3. Table 3 reports the training time on each dataset.

Table 3: Model Training Time.

Dataset	Training Time (CNN-LSTM)	Inference Time (per sample)	XGBoost Training Time
CM1	~7 min	~0.8 ms	~15 sec
KC1	~9 min	~0.9 ms	~20 sec
PC1	~10 min	~0.95 ms	~18 sec
PC4	~12 min	~1.0 ms	~22 sec
MC1	~15 min	~1.1 ms	~25 sec

Evaluation Metrics:

The five-performance metrics used in the study to assess the performance of the developed model were Accuracy, precision, recall, F1 score, and AUC-ROC. In the relevant research, these metrics are frequently used to assess SDP performance.

The following metrics are calculated in equations 12, 13, 14, 15, and 16.

$$\textbf{Accuracy} = \frac{\mathrm{M+P}}{\mathrm{M+P+O+N}} \quad (12)$$

$$\textbf{Recall} = \frac{\mathrm{M}}{\mathrm{M+N}} \quad (13)$$

$$\textbf{Precision} = \frac{\mathrm{M}}{\mathrm{M+O}} \quad (14)$$

$$\textbf{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision+Recall}} \quad (15)$$

$$\boldsymbol{AUC} = \sum_{i-1}^{n-1} \frac{(Q_{i+1} - Q_i) \times (R_{i+1} + R_i}{2} \quad (16)$$

Where:

True Positive (M): when a defective instance is accurately identified as defects.

True Negative (P): when a non-defective instance is accurately identified as non-defects.

False Positive (Q): when a non-defective instance is mistakenly identified as defects.

False Negative (P): when a defective instance is mistakenly identified as non-defects.

Experimental Setup:

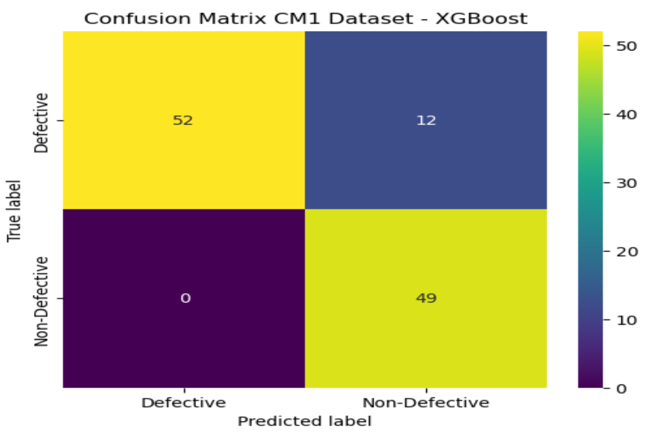
Python 3.9 was used for the study, along with well-known libraries including Matplotlib, Scikit-learn, and TensorFlow. An Apple Silicon processor, 16GB of RAM, and an 8 Core GPU for faster calculations were all part of the hardware setup.

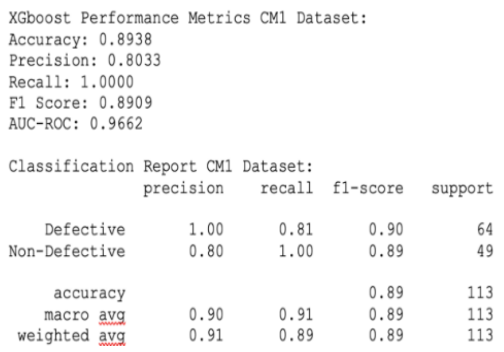
4. RESULTS AND DISCUSSION

This section discussed the XGboost model's performance in predicting software defects on five datasets. ROC Curves and

Confusion matrix were additionally used to determine the findings. It also talks about how effective the hybrid technique is. However, the proposed model demonstrates strong overall performance across all datasets, although some variation in accuracy is observed between them. For instance, the CM1 dataset achieved an accuracy of 0.8938, while the PC4 dataset reached 0.9918.

This difference can be attributed to several factors like Class imbalance which CM1 has fewer defective instances even after applying ADASYN, making defect detection more challenging. Dataset size of CM1 contains only 327 instances, limiting the model's learning capability compared to larger datasets like PC4. And also, feature relevance and noise with some datasets may contain less informative or noisier features, affecting prediction accuracy. Despite these variations, the model consistently achieves high recall and AUC-ROC scores across all datasets, indicating robust defect discrimination ability. These results suggest that while the model performs exceptionally well on balanced and larger datasets, it remains effective for smaller or imbalanced codebases, especially in terms of minimizing missed defects. This supports the generalizability of the proposed method across different software environments, provided that sufficient and representative data are available.

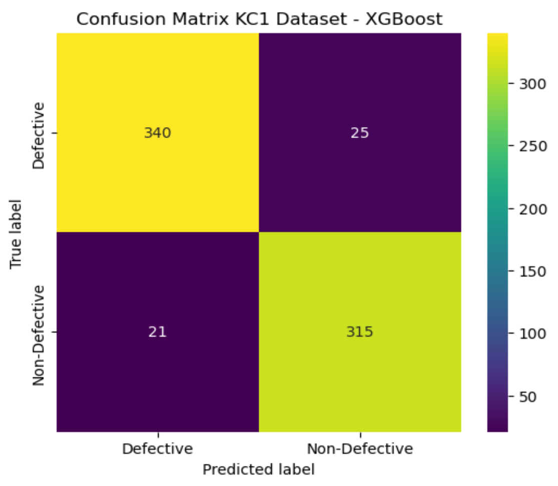


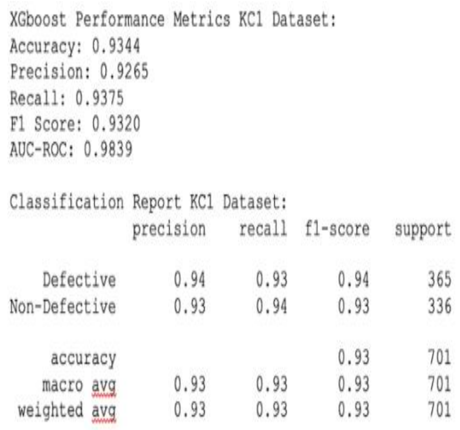


```
XGboost Performance Metrics CM1 Dataset:
Accuracy: 0.8938
Precision: 0.8033
Recall: 1.0000
F1 Score: 0.8909
AUC-ROC: 0.9662

Classification Report CM1 Dataset:
               precision    recall  f1-score   support

    Defective       1.00      0.81      0.90        64
Non-Defective       0.80      1.00      0.89        49

     accuracy                           0.89       113
    macro avg       0.90      0.91      0.89       113
 weighted avg       0.91      0.89      0.89       113
```

Figure 4: CM1 Dataset Confusion Matrix

The proposed approach works well on the CM1 dataset, achieving accuracy, precision, recall, F1-score of 0.8938, 0.8033, 1.0000, 0.8909, respectively and AUC-ROC of 0.9662. The perfect recall for defective modules ensures no potential defects are missed, while the high AUC-ROC value highlights the model's effective discrimination between classes. However, the precision of 0.8033 indicates a trade-off with false positives, as 20 non-defective modules were incorrectly classified as defective. The confusion matrix in Figure 4 reveals 49 true positives out of 64 actual defective modules and 52 true negatives out of 49 non-defective modules, showcasing the model's ability to minimize false negatives but pointing to an opportunity for reducing false positives. The hybrid approach offers superior performance in terms of recall and AUC-ROC while maintaining computational efficiency.

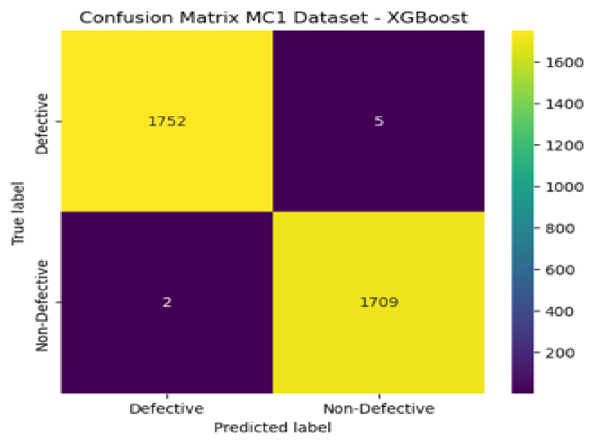


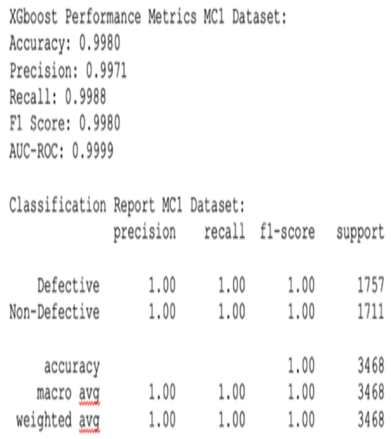


```
XGboost Performance Metrics KC1 Dataset:
Accuracy: 0.9344
Precision: 0.9265
Recall: 0.9375
F1 Score: 0.9320
AUC-ROC: 0.9839

Classification Report KC1 Dataset:
               precision    recall  f1-score   support

    Defective       0.94      0.93      0.94       365
Non-Defective       0.93      0.94      0.93       336

     accuracy                           0.93       701
    macro avg       0.93      0.93      0.93       701
 weighted avg       0.93      0.93      0.93       701
```

Figure 5: KC1 Dataset Confusion Matrix

With an accuracy, precision, recall, and F1-score of 0.9344, 0.9265, 0.9375, and 0.9320, respectively, and an AUC-ROC of 0.9839, the proposed approach performs well on the KC1 dataset. As seen by the high AUC-ROC value, these results demonstrate the model's efficacy in differentiating between defect and non-defect modules. Defective modules earn 0.94, 0.93, and F1-score, whereas non-defective modules achieve 0.93, 0.94, and F1-score, respectively, according to the classification report, which shows balanced performance across both classes. This balanced performance indicates the model effectively reduces incorrect classifications, that is to say, both false positives, where non-defective items get flagged wrongly, and false negatives, where actual issues go undetected, while delivering consistent outcomes across categories. These findings gain further support from the confusion matrix shown in Figure 5, which demonstrates that among 365 defective modules, 340 were identified correctly true positives, to put it simply whereas 25 slipped through as non-defective cases, what we call false negatives. Similarly, the model achieved notable true negative rates by mistakenly labelling only 21 out of 336 defect-free modules as problematic, known as false positives. When evaluated against the KC1 dataset, the combined approach shows accuracy levels matching those of standalone deep learning systems while outperforming conventional machine learning models in overall performance metrics, illustrating its strength in detecting complex data patterns without demanding excessive computational resources, so to speak. This balanced approach proves particularly valuable given how it maintains operational efficiency while handling intricate classification tasks, a crucial consideration given the real-world constraints often present in such implementations. The findings demonstrate the hybrid approach's value as a practical, real-world tool for enhancing software quality by confirming its robustness and dependability in SDP.

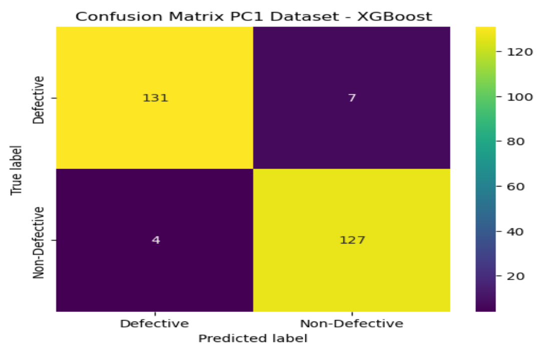


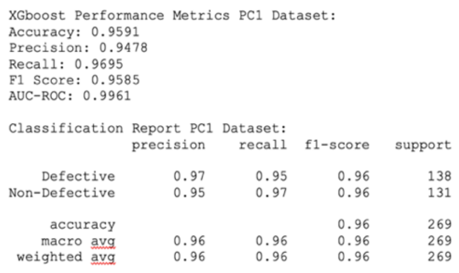


```
XGboost Performance Metrics MC1 Dataset:
Accuracy: 0.9980
Precision: 0.9971
Recall: 0.9988
F1 Score: 0.9980
AUC-ROC: 0.9999

Classification Report MC1 Dataset:
              precision    recall  f1-score   support

   Defective       1.00      1.00      1.00      1757
Non-Defective      1.00      1.00      1.00      1711

    accuracy                           1.00      3468
   macro avg       1.00      1.00      1.00      3468
weighted avg       1.00      1.00      1.00      3468
```

Figure 6: MC1 Dataset Confusion Matrix

With an accuracy, precision, recall, and F1-score of 0.9980, 0.9971, 0.9988, and 0.9980, respectively, and an AUC-ROC of 0.9999, the hybrid approach performs exceptionally well on the MC1 dataset. As evidenced by the nearly flawless AUC-ROC score, these results show the model's remarkable capacity to discriminate between defective and non-defective modules. With precision, recall, and F1-scores of 1.00 for both defect and non-defect modules, the classification report shows almost perfect performance for both classes. The recall was 1.00, with only 5 false negatives. Specifically, 1752 of the 1757 real problematic modules were accurately discovered (true positives). A precision of 1.00 and a recall of 1.00 were obtained by misclassifying only 2 out of 1711 non-defective modules as defective (false positives). The model's excellent accuracy is further supported by the confusion matrix in Figure 6, which displays few errors: only five defective modules were mistakenly categorized as non-defective, while two non-defective modules were mistakenly classified as defective. The hybrid technique performs better on the MC1 dataset than either standalone deep learning approaches or ML models, especially in terms of recall and AUC-ROC, while still being computationally efficient. The proposed approach is highly suitable for practical software defect prediction applications due to its exceptional accuracy and balanced performance between recall and precision metrics. These results validate the hybrid approach's effectiveness in providing accurate and dependable predictions, making a substantial contribution to the advancement of software defect identification techniques.



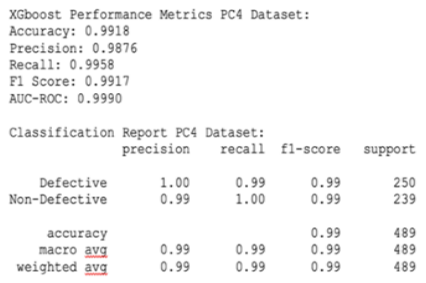


```
XGboost Performance Metrics PC1 Dataset:
Accuracy: 0.9591
Precision: 0.9478
Recall: 0.9695
F1 Score: 0.9585
AUC-ROC: 0.9961

Classification Report PC1 Dataset:
               precision    recall  f1-score   support

    Defective       0.97      0.95      0.96       138
Non-Defective       0.95      0.97      0.96       131

     accuracy                           0.96       269
    macro avg       0.96      0.96      0.96       269
 weighted avg       0.96      0.96      0.96       269
```

Figure 7: PC1 Dataset Confusion Matrix

The hybrid approach performs exceptionally well on the PC1 dataset, with accuracy, precision, recall, and F1-score of 0.9591, 0.9478, 0.9695, and 0.9585, respectively, and an AUC-ROC of 0.9961. Given the high AUC-ROC value, these results demonstrate the

odel's great ability to discriminate between defective and non-defective modules. The classification report shows that both classes perform equally, with non-defective modules obtaining a precision of 0.95, a recall of 0.97, and an F1-score of 0.96, and defective modules obtaining a precision of 0.97, a recall of 0.95, and an F1-score of 0.96. To put it simply, this balanced state indicates the model successfully reduces both incorrect rejections and false alarms. These findings align with the confusion matrix shown in Figure 7, which reveals that 131 out of 138 actual defective cases were correctly identified, achieving a detection rate of 0.95 while only missing 7 instances. The precision and recall were 0.95 and 0.97, respectively, with just 4 out of 131 non-defective modules being incorrectly categorized as defective (false positives). On the PC1 dataset, the hybrid technique demonstrates its efficacy in collecting intricate patterns while preserving computing economy by achieving competitive accuracy and higher AUC-ROC when compared to independent deep learning approaches or traditional machine learning models. The findings validate the hybrid approach's effectiveness and dependability for SDP, establishing it as a valuable resource for improving software quality in real-world applications.



```
XGboost Performance Metrics PC4 Dataset:
Accuracy: 0.9918
Precision: 0.9876
Recall: 0.9958
F1 Score: 0.9917
AUC-ROC: 0.9990

Classification Report PC4 Dataset:
               precision    recall  f1-score   support

    Defective       1.00      0.99      0.99       250
Non-Defective       0.99      1.00      0.99       239

     accuracy                           0.99       489
    macro avg       0.99      0.99      0.99       489
 weighted avg       0.99      0.99      0.99       489
```

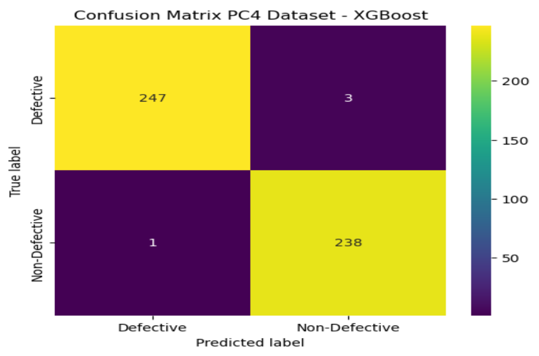


Figure 8: PC4 Dataset Confusion Matrix

The proposed approach demonstrates exceptional performance on the PC4 dataset, achieving an accuracy, precision, recall, F1-Score of 0.9918, 0.9876, 0.9958, 0.9917, respectively and AUC-ROC of 0.9990. The near-perfect AUC-ROC value highlights the model's outstanding ability to distinguish between defective and non-defective modules. The classification report further confirms the model's robustness, with precision, recall, and F1-scores of approximately 0.99 for both defective and non-defective classes. In particular, 247 of the 250 real problematic modules were effectively detected (true positives), yielding a 0.99 recall and few false negatives (3). Similarly, among 239 non-defective modules, only 1 was misclassified, yielding a precision of 0.99 and a recall of 1.00. Figure 8 confusion matrix supports these results, demonstrating the model's high accuracy with minimal errors: only 3 defective modules were misclassified as non-defective, and just 1 non-defective module was incorrectly identified as defective in the PC4 dataset, the hybrid technique outperforms standalone deep learning approaches or classical machine learning models, especially in terms of recall and AUC-ROC, while retaining computational efficiency. Because of its high accuracy and ability to balance precision and recall, the proposed approach is ideal for real-world SDP applications.

Comparison with State of Art Models:

In this section, Table 4 compares the proposed technique to different present techniques using the performance metrics used for the study.

It should be noted that some of the methods compared in Table 4 were evaluated on different datasets (e.g., PROMISE, bug report datasets, GHPR). These datasets differ in terms of size, language, and feature composition, which can impact model performance. Therefore, while the comparison provides a general sense of the proposed method's effectiveness relative to existing approaches, it should be interpreted with caution due to the lack of uniformity in data sources.

Table 4: Comparison with State of Art models

Techniques	**Datasets**	**Accuracy**	**Precision**	**Recall**	**F1 Score**	**AUC**
CBIL model (Farid *et al*., 2021)	PROMISE datasets (Jedit, Lucene, Synapse, Xalan, Xerces, Camel, Poi)					0.91, 0.83, 0.95, 0.76, 0.98, 0.96, 0.95
CNN and Random Forest with Boosting (Kukkar *et al*., 2019)	Bug report datasets (JBoss, Eclipse, Open FOAM, Firefox, Mozilla,)	0.98, 0.95, 0.94, 0.97, 0.94				
(CNN, BI- LSTM) (Khleel & Nehéz, 2022)	GHPR Dataset	0.80, 0.80	0.77 ,0.77	0.84, 0.85	0.81, 0.80	0.83, 0.84
VESDP (Ali *et al*., 2024)	CM1, JM1, MC2, PC1, PC4	86.87, 79.92, 68.42, 92.16, 87.14				
XGboost+ Under sampling (AL-Hadidi & Hasoon, 2024)	CM1, MC1, KC1, PC1, PC4	0.88, 0.97, 0.74, 0.93, 0.93	0.84, 0.95, 0.75, 0.92, 0.92	0.93, 0.99, 0.72, 0.94, 0.96	0.88, 0.97, 0.74, 0.93, 0.94	-
ADAYSN+ CNN + LSTM XGboost (This Study)	CM1, MC1, KC1, PC1, PC4	**0.89, 0.99, 0.93, 0.95, 0.99**	**0.80, 0.99, 0.92, 0.94, 0.98**	**1.0, 0.99, 0.93 ,0.95, 0.99**	**0.89, 0.99, 0.93, 0.95, 0.99**	**0.96, 0.99, 0.98, 0.99, 0.99**

A comprehensive analysis of different software defect prediction techniques across several datasets is given in Table 3, which assesses each method's performance using metrics like accuracy, precision, recall, F1 score, and AUC. Strong AUC values between 0.76 and 0.98 are reported by the CBIL model (Farid *et al*., 2021), which focusses on PROMISE datasets and shows good discriminative capability. The lack of additional measures, however, makes it more difficult to evaluate its overall efficacy. Similar to this, the CNN and Random Forest with Boosting approach (Kukkar *et al*., 2019), which was tested on bug report datasets from JBoss, Eclipse, OpenFOAM, Firefox, and Mozilla, achieves high accuracy values (0.94–0.98) but is hard to compare with other methods because it does not provide information on precision, recall, F1 score, or AUC. Using the GHPR dataset, the CNN + BI-LSTM approach (Khleel & Nehéz, 2022), shows balanced performance with accuracy values of 0.80, precision around 0.77, recall between 0.84 and 0.85, and F1 scores near 0.81. Its moderate to strong classification skill is reported by its AUC values, which range from 0.83 to 0.84. On the other hand, accuracy values for the VESDP approach [24], which was evaluated on the CM1, JM1, MC2, PC1, and PC4 datasets, range from 68.42% to 92.16%. However, a thorough assessment of its effectiveness is hampered by the absence of precision, recall, F1 score, and AUC measures. The XGboost + Under Sampling approach (AL-Hadidi & Hasoon, 2024) shows competitive results on CM1, MC1, KC1, PC1, and PC4 datasets (refer to Figure 9), achieving accuracy values of 0.88–0.97, precision ranging from 0.75–0.95, and recall between 0.72–0.99. While it performs well in terms of F1 scores (0.88–0.97), the absence of AUC values leaves a gap in understanding its classification capability. The proposed ADASYN + CNN + LSTM + XGboost technique stands out by delivering superior performance across all metrics. Evaluated on the same datasets (CM1, MC1, KC1, PC1, and PC4), it achieves accuracy values of 0.89–0.99, precision ranging from 0.80–0.99, near-perfect recall (1.0–0.99), and F1 scores between 0.89–0.99. Additionally, the AUC values remain notably high (0.96–0.99), demonstrating strong discriminatory power between defective and non-defective cases, that is to say, showing the model's effectiveness in differentiation. The combination of balancing techniques for dataset categories with feature extraction methods and classification approaches effectively addresses issues like uneven data distribution and complex pattern recognition.

The currently available methods actually show some good results in certain specific areas, but this study's hybrid approach clearly provides a much more comprehensive solution to predict software defects. By combining both deep learning along with more traditional machine learning techniques, the approach ends up achieving excellent performance across many different metrics. Additionally, the method shows particularly strong results in both recall and AUC, especially when dealing with uneven datasets like MC1 and PC4. Because of its improved accuracy and reliability, this new method has become particularly valuable for eventually enhancing software quality in many real-world applications if deployed.

However, several state-of-the-art methods compared in Table 3 were evaluated on different datasets or reported only partial metrics (e.g., accuracy or AUC-ROC without corresponding precision, recall, or F1-scores). These inconsistencies limit the depth of direct comparisons. However, based on the available data, the proposed hybrid method demonstrates consistently strong performance across all reported metrics and datasets, particularly in terms of recall and AUC-ROC, which are crucial in imbalanced software defect prediction tasks.

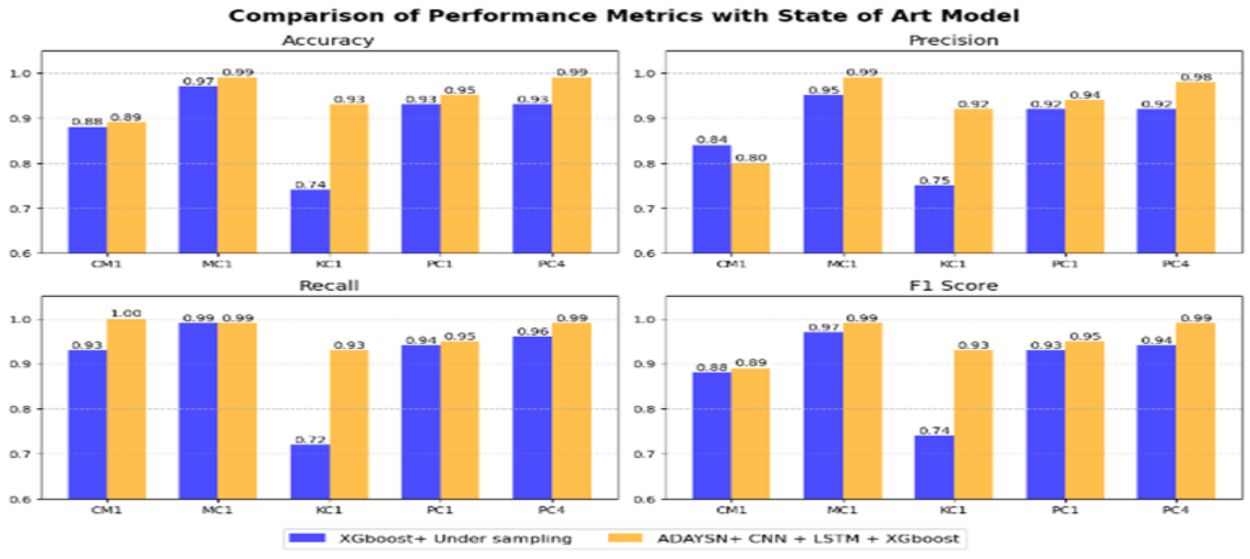


Figure 9: Comparative performance with (AL-Hadidi & Hasoon, 2024)

CONCLUSION

The hybrid CNN-LSTM model with ADASYN balancing and XGboost training actually ends up providing a much more effective approach to software defect prediction. This particular method clearly helps to strengthen the overall software quality because it fully addresses many class imbalance issues and eventually helps detect more complex patterns in software metrics that might otherwise just go unnoticed.

Using the benchmark dataset of CM1, KC1, PC1, PC4, and MC1. The model achieves high performance in key evaluation metrics, with average accuracy of 0.958 recall of 0.993, and AUC-ROC of 0.987, significantly outperforming traditional machine learning and standalone deep learning methods. Notably, it achieves perfect recall on the CM1 dataset and near-perfect AUC-ROC scores across all datasets, highlighting its ability to detect defective modules with minimal false negatives, which is crucial in real-world applications. The testing currently shows that the hybrid approach performs much better than many traditional machine learning methods, delivering more improved accuracy and still maintains resilience in predictive modelling. Additionally, the method's effectiveness is fully demonstrated through comparative analysis with some current models, particularly in terms of the AUC-ROC and recall metrics. Moreover, further validation across different types of software systems and some exploration of optimization techniques would help to establish this method's broader applicability in the field of software engineering. These important innovations in software defect prediction ultimately provide a more comprehensive solution that effectively addresses many of the challenges that are currently faced in modern software development. To further validate the observed improvement, future work should include statistical significance testing, like the Wilcoxon signed-rank test or bootstrapping approaches, to guarantee the performance gains are not random. A more detailed ablation study is recommended to quantify the impact of each component on performance metrics such as recall and AUC-ROC. This would provide deeper insight into the effectiveness of the hybrid architecture and guide future model optimization. Furthermore, exploring hyperparameter optimization, model interpretability using XAI techniques, and cross-project generalization can improve both the practical utility and theoretical understanding of the model. This study provides a reliable and scalable solution for software defect prediction, offering valuable insights into how hybrid deep learning and machine learning methods can be integrated to improve predictive accuracy and assist automated quality assurance in software development.

Acknowledgments:

Special appreciation goes to the Biology Department at the University of Zakho for providing the necessary facilities and academic environment.

Author Contributions:

The authors conducted all parts of this article.

Ethical Approval:

This research did not require ethical approval because it did not involve humans or animals.

Funding:

None.

REFERENCES

Ahmed, S. F., Alam, M. S. B., Hassan, M., Rozbu, M. R., Ishtiak, T., Rafa, N., & Gandomi, A. H. (2023). Deep Learning Modelling Techniques: Current Progress, Applications, Advantages, and Challenges. *Artificial Intelligence Review*, *56*(11), 13521–13617. https://doi.org/10.1007/s10462-023-10409-2

AL-Hadidi, T. N., & Hasoon, S. O. (2024). Software Defect Prediction Using Extreme Gradient Boosting (XGBoost) with Optimization Hyperparameter. *Al-Rafidain Journal of Computer Sciences and Mathematics (RJCM)*, *18*(1), 22–29. https://doi.org /10.33899/csmj.2023.142739.1081

Ali, M., Mazhar, T., Arif, Y., Al-Otaibi, S., Ghadi, Y. Y., Shahzad, T., Khan, M. A., & Hamam, H. (2024). Software Defect Prediction Using an Intelligent Ensemble-Based Model. *IEEE Access*. https://doi.org/10.1109/ACCESS.2024.3358201

Alkaberi, W., & Assiri, F. (2024). Predicting the Number of Software Faults using Deep Learning. *Engineering, Technology &amp; Applied Science Research*, *14*(2), 13222–13231. https://doi.org/10.48084/etasr.6798

Alzeyani, E. M. M., & Szabó, C. (2024). Comparative Evaluation of Model Accuracy for Predicting Selected Attributes in Agile Project Management. *Mathematics*, *12*(16), 2529. https://doi.org/10.3390/math12162529

Charles, J. (2024). Revolutionizing Software Project Development: A CNN-LSTM Hybrid Model for Effective Defect Prediction. *International Journal of Advanced Computer Science & Applications*, *15*(1). https://doi.org/10.14569/ijacsa.2024.0150158

Elentukh, A. (2023). People Make Mistakes–A Survey of Common Causes of Software Defects. *International Conference on Computer Science and Education in Computer Science*, 117–133. https://doi.org/10.1007/978-3-031-44668-9_9

Farabet, C., Couprie, C., Najman, L., & LeCun, Y. (2013). Learning Hierarchical Features for Scene Labeling. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, *8*, 1915–1929. https://doi.org/10.1109/TPAMI.2012.231

Farabet, C., Couprie, C., Najman, L., & LeCun, Y. (2013). Learning Hierarchical Features for Scene Labeling. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, *8*, 1915–1929. https://doi.org/10.1109/TPAMI.2012.231

Farid, A. B., Fathy, E. M., Eldin, A. S., & Abd-Elmegid, L. A. (2021). Software Defect Prediction Using Hybrid Model (CBIL) of Convolutional Neural Network (CNN) and Bidirectional Long Short-Term Memory (Bi-LSTM). *PeerJ Computer Science*, *7*, e739. https://doi.org/10.7717/peerj-cs.739

Feyzi, F., & Daneshdoost, A. (2023). Studying the effectiveness of deep active learning in software defect prediction. *International Journal of Computers and Applications*, *45*(7–8), 534–552. https://doi.org/10.1080/1206212X.2023.2252117

Ghotra, B., McIntosh, S., & Hassan, A. E. (2017). A large-scale study of the impact of feature selection techniques on defect classification models. *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 146–157. https://doi.org/10.1109/MSR.2017.18

Giray, G., Bennin, K. E., Köksal, Ö., Babur, Ö., & Tekinerdogan, B. (2023). On the use of deep learning in software defect prediction. *Journal of Systems and Software*, *195*, 111537. https://doi.org/10.1016/j.jss.2022.111537

Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep feedforward networks. *Deep learning*, *1*, 161-217.

Goyal, S. (2022). Handling class-imbalance with KNN (neighbourhood) under-sampling for software defect prediction. *Artificial Intelligence Review*, *55*(3), 2023–2064. https://doi.org/10.1007/s10462-021-10044-w

Hussein, A. S., Li, T., Abd Ali, D. M., Bashir, K., & Yohannese, C. W. (2020). A modified adaptive synthetic sampling method for learning imbalanced datasets. *Developments of Artificial Intelligence Technologies in Computation and Robotics: Proceedings of the 14th International FLINS Conference (FLINS 2020)*, 76–83. https://doi.org/10.1142/9789811223334_0010

Jin, C. (2021). Cross-project software defect prediction based on domain adaptation learning and optimization. *Expert Systems with Applications*, *171*. https://doi.org/10.1016/j.eswa.2021.114637

Jude, A., & Uddin, J. (2024). Explainable Software Defects Classification Using SMOTE and Machine Learning. *Annals of Emerging Technologies in Computing (AETiC)*, *8*(1). https://doi.org/10.33166/AETiC.2024.01.00

Khalid, A., Badshah, G., Ayub, N., Shiraz, M., & Ghouse, M. (2023). Software Defect Prediction Analysis Using Machine Learning Techniques. *Sustainability*, *15*(6). https://doi.org/10.3390/su15065517

Khan, M. F. I., & Masum, A. K. M. (2024). Predictive Analytics and Machine Learning for Real-Time Detection of Software Defects and Agile Test Management. *Educational Administration: Theory and Practice*, *30*(4), 1051–1057.

Khleel, N. A. A., & Nehéz, K. (2022). A New Approach to Software Defect Prediction Based on Convolutional Neural Network and Bidirectional Long Short-Term Memory. *Production Systems and Information Engineering*, *10*(3), 1–18. https://doi.org/10.32968/psaie.2022.3.1

Krasner, H. (2021). The cost of poor software quality in the US: A 2020 report. *Proc. Consortium Inf. Softw. QualityTM (CISQTM)*, *2*.

Kukkar, A., Mohana, R., Nayyar, A., Kim, J., Kang, B. G., & Chilamkurti, N. (2019). A Novel Deep-Learning-Based Bug Severity Classification Technique Using Convolutional Neural Networks and Random Forest with Boosting. *Sensors*, *19*(13), 2964. https://doi.org/10.3390/s19132964

Kumar, L., Singh, V., Neti, L. B. M., Misra, S., & Krishna, A. (2023). An Empirical Framework for Software Aging-Related Bug Prediction using Weighted Extreme Learning Machine. *FedCSIS (Communication Papers)*, 181–188. https://doi.org/10.15439/2023F9248

Maddipati, S., & Srinivas, M. (2021). A Hybrid Approach for Cost Effective Prediction of Software Defects. *International Journal of Advanced Computer Science and Applications*, *12*. https://doi.org/10.14569/IJACSA.2021.0120219

Mahmoud, A. N., Abdelaziz, A., Santos, V., & Freire, M. M. (2024). A proposed model for detecting defects in software projects. *Indonesian Journal of Electrical Engineering and Computer Science*, *33*(1), 290–302. https://doi.org /10.11591/ijeecs.v33.i1

Mehmood, I., Shahid, S., Hussain, H., Khan, I., Ahmad, S., Rahman, S., Ullah, N., & Huda, S. (2023). A Novel Approach to Improve Software Defect Prediction Accuracy Using Machine Learning. *IEEE Access*. https://doi.org /10.1109/ACCESS.2023.3287326

Menzies, T., Krishna, R., & Pryor, D. (2015). *The Promise Repository of Empirical Software Engineering Data*. http://openscience.us/repo

Nevendra, M., & Singh, P. (2022). A survey of software defect prediction based on deep learning. *Archives of Computational Methods in Engineering*, *29*(7), 5723–5748. https://doi.org/10.1007/s11831-022-09787-8

Olaleye, T. O., Arogundade, O. T., Misra, S., Abayomi-Alli, A., & Kose, U. (2023). Predictive analytics and software defect severity: A systematic review and future directions. *Scientific Programming*, *2023*(1), 6221388. https://doi.org/10.1155/2023/6221388

Olorunshola, O. E., Irhebhude, M. E., Evwiekpaefe, A. E., & Ogwueleka, F. N. (2020). Evaluation of machine learning classification techniques in predicting software defects. *Trans. Mach. Learn. Artif. Intel*, *8*, 1–15.

Pachouly, J., Ahirrao, S., Kotecha, K., Selvachandran, G., & Abraham, A. (2022). A systematic literature review on software defect prediction using artificial intelligence: Datasets, Data Validation Methods, Approaches, and Tools. *Engineering Applications of Artificial Intelligence*, *111*, 104773.https://doi.org/10.1016/j.engappai.2022.104773

Patil, D., Rane, N. L., Desai, P., & Rane, J. (2024). Machine learning and deep learning: Methods, techniques, applications, challenges, and future research opportunities. *Trustworthy Artificial Intelligence in Industry and Society*, 28–81. https://doi.org/10.70593/978-81-981367-4-9

Ponnala, R., & Reddy, C. (2023). Ensemble Model for Software Defect Prediction Using Method Level Features of Spring Framework Open Source Java Project for E-Commerce. *Shu Ju Cai Ji Yu Chu Li/Journal of Data Acquisition and Processing*, *38*, 1645–1650. https://doi.org/10.5281/zenodo.7749985

Saidani, I., Ouni, A., & Mkaouer, M. W. (2022). Improving the prediction of continuous integration build failures using deep learning. *Automated Software Engineering*, *29*(1), 21. https://doi.org/10.1007/s10515-021-00319-5

Shafiq, M., Alghamedy, F. H., Jamal, N., Kamal, T., Daradkeh, Y. I., & Shabaz, M. (2023). Retracted: Scientific programming using optimized machine learning techniques for software fault prediction to improve software quality. *IET Software*, *17*(4), 694–704. https://doi.org/10.1155/2020/8858010

Shen, Z., & Chen, S. (2020). A survey of automatic software vulnerability detection, program repair, and defect prediction techniques. *Security and Communication Networks*, *2020*(1), 8858010. https://doi.org/10.1155/2020/8858010

Shepperd, M., Song, Q., Sun, Z., & Mair, C. (2018). *NASA MDP Software Defects Data Sets*. figshare. https://doi.org/10.6084/m9.figshare.c.4054940.v1

Tameswar, K., Suddul, G., & Dookhitram, K. (2022). A hybrid deep learning approach with genetic and coral reefs metaheuristics for enhanced defect detection in software. *International Journal of Information Management Data Insights*, *2*(2), 100105. https://doi.org/10.1016/j.jjimei.2022.100105

Thomas, N. S., & Kaliraj, S. (2024). An Improved and Optimized Random Forest Based Approach to Predict the Software Faults. *SN Computer Science*, *5*(5), 530. https://doi.org/10.1007/s42979-024-02764-x

Vogel-Heuser, B., Fay, A., Schaefer, I., & Tichy, M. (2015). Evolution of software in automated production systems: Challenges and research directions. *Journal of Systems and Software*, *110*, 54–84. https://doi.org/10.1016/j.jss.2015.08.026

Wan, X., Zheng, Z., Qin, F., & Lu, X. (2024). Data complexity: A new perspective for analyzing the difficulty of defect prediction tasks. *ACM Transactions on Software Engineering and Methodology*. https://doi.org/10.1145/3649596

Wang, D., Zhang, B., & Zhu, M. (2022). A survey on convolutional neural network with its applications. *Comput. Math. Appl.*, *83*(3), 186–206. https://doi.org/10.1016/j.camwa.2021.11.025

Wang, H., Zhuang, W., & Zhang, X. (2021). Software defect prediction based on gated hierarchical LSTMs. *IEEE Transactions on Reliability*, *70*(2), 711–727. https://doi.org/10.1109/TR.2020.3047396

Wang, S., Huang, L., Gao, A., Ge, J., Zhang, T., Feng, H., Satyarth, I., Li, M., Zhang, H., & Ng, V. (2022). Machine/deep learning for software engineering: A systematic literature review. *IEEE Transactions on Software Engineering*, *49*(3), 1188–1231. https://doi.org/10.1109/TSE.2022.3173346