OPEN ACCESS

ISAR Journal of Science and Technology

Volume 4, Issue 2, 2026, Page: 1-11

Abbriviate Title- ISAR J Sci Tech

ISSN *(Online)*- 2584-2056

https://isarpublisher.com/journal/isarjst

A Hybrid Neuro-Fuzzy Machine Learning Framework for Predicting Software Quality in Open-Source Projects

Siman Emmanuel[1*], Ofubu Vivian Elawen[2], Moshood A. Hambali[3], Agu Edward Onyebueke[4], Madugu Jeremiah Omanga[5], and Ajiboye Oluwatimilehin Jonathan[6]

[1,2,3]Faculty of Engineering, University Technology Malaysia, Skudai 81310, Malaysia

[1]Federal University Wukari, PMB 1020 Wukari, Taraba State.

***Corresponding Author**

Siman Emmanuel

Faculty of Engineering, University Technology Malaysia, Skudai 81310, Malaysia.

Article History

Received: 22.11.2025
Accepted: 02.01.2026
Published: 09.02.2026

Abstract: The increasing complexity of software systems, especially in Free and Open-Source Software (FOSS) ecosystems, there has been a growing requirement for the adoption of automated software quality prediction tools. In this context, this study attempts to present a hybrid Neuro-Fuzzy Machine Learning (NF-ML) model to predict software quality by identifying defective software modules. In this study, the proposed model combines the efficient learning capability offered by neural networks and the malleable reasoning capabilities offered by fuzzy logic to create a robust model with high interpretability. In this experiment, the NASA Metrics Data Program (MDP) dataset sourced from Kaggle was used to train and test the proposed model. Additionally, data pre-processing was conducted to address missing data points and normalize the data points using chi-squared feature selection. The proposed model was developed using TensorFlow programming from Python with various performance metrics such as accuracy, precision, recall, F1-measure, and AUC-ROC accuracy to determine efficiency. The experimental outcome of this research study suggests that the hybrid NF-ML model attains better efficiency than ML models to predict software quality while increasing reliability.

Keywords: Neuro-fuzzy systems; machine learning; software quality prediction; defect detection; fuzzy logic; neural networks; Free and Open-Source Software (FOSS); software reliability.

Cite this article:

Emmanuel, S., Elawen, O. V., Hambali, M. A., Onyebueke, A. E., Omanga, M. J., and Jonathan, A. O., (2026). A Hybrid Neuro-Fuzzy Machine Learning Framework for Predicting Software Quality in Open-Source Projects. *ISAR Journal of Science and Technology*, *4*(2), 1-11.

1. Introduction

Software quality prediction has become a cornerstone of software engineering research caused by the increasing size and complexity of modern software systems. In Free and Open-Source Software (FOSS) projects, quality control poses unique challenges owing to decentralized development, varying coding standards, and diverse contributor skill levels. Conventional software quality assurance techniques, largely dependent on manual inspection and rule-based testing, often fail to detect defects early and efficiently. Machine learning (ML) and fuzzy logic are powerful tools for predictive software analytics. Neural networks excel in adaptive learning, while fuzzy logic provides interpretability and the ability to model uncertainty. However, standalone ML or fuzzy logic systems have limitations in handling complex, nonlinear relationships among software metrics. Hence, hybrid models such as Neuro-Fuzzy systems have gained attention for combining the learning capacity of neural networks with the rule-based reasoning of fuzzy systems. This study presents a hybrid Neuro-Fuzzy Machine Learning framework aimed at predicting software quality in open-source environments. The framework leverages data-driven learning while preserving interpretability through fuzzy inference, providing a more transparent and accurate prediction mechanism.

2. Related Work

In 2015, Arar & Ayan presented cost-sensitive software defect prediction using a neural network. Internal metrics were considered in the research. The code in software systems was taken into

account, but not their functionality. An alternative method of classification was introduced in the paper. Conventional Artificial Neural Networks were used together with a new Artificial Bee Colony algorithm in the paper. In 2017, Li et al. introduced a novel method of software defect prediction using Convolutional Neural Networks. Conventional methods of defect prediction were highly dependent on hand-crafted features. Roy et al., (2019), introduced the forecasting of software reliability through the usage of the neighborhood fuzzy particle swarm optimization on the basis of the new neural network approach. The research study introduced the artificial neural network (ANN) model for software reliability, which was trained using the new particle swarm optimization (PSO) method for accurate forecasting of the software reliability. The new proposed ANN model was introduced after considering the generation of the fault phenomenon in the process of software testing depending on the fault complexity at different levels. Moreover, the authors also introduced the new neighborhood fuzzy particle swarm optimization method for proper learning of the proposed method through the software failure data. Afterwards, the study by Qiao et al., (2020) proposed a new methodology that uses deep learning to deal with the increasing complexity of software systems, thereby making it difficult to prevent software defects. The study stresses the need to correctly forecast the rates of software module defects to make efficient allocation of resources to programmers. Zhen et al., in 2020, proposed a hybrid approach based on the optimization capabilities of the Wolf Pack Algorithm (WPA) and Particle Swarm Optimization technique to enhance software reliability prediction results. The research gaps identified are addressed by introducing a hybrid approach between WPA and PSO, and then developing a fitness function by applying the Maximum Likelihood Estimation technique to estimate parameters in software reliability equations. When it comes to predicting defects in software, an interesting approach was presented by Nevendra & Singh (2021) using improved Convolutional Neural Networks (CNNs). The experimental results show a marked increase in performance compared to some existing approaches, such as Convolutional Neural Network (CNNs) and classical machine learning methods. Kumar *et al*. (2022) presented a novel three-phase model aimed at improving the prediction of software component reusability within the context of component-based software development. The proposed model significantly outperforms traditional HHO algorithms and other comparative models, achieving an approximate 2% improvement in similarity values and demonstrating superior accuracy and effectiveness in predicting software component reusability.

Ingrid et al (2023) performed bug prediction analysis on two software projects," commons-cli" and" commons-compress," employing different machine learning algorithms and an ensemble method, Voting Classifier. The outcome yielded high AUC values. However, the Voting Classifier led to a considerable improvement in the accuracy of bug prediction compared to individual machine learning algorithms. The software defect prediction model proposed by Eldamarany (2023) provided a reliable solution to software defect prediction by integrating effective machine learning algorithms with interpretability methods. The findings of this paper are beneficial to practitioners and researchers seeking to improve software quality, especially within the context of Free and Open-Source Software. Sharma *et al.* (2023) introduced a comprehensive framework for defect prediction using eleven classifiers across twelve datasets. They tackled class imbalance using SMOTE and emphasized model transparency through Shapley values. Their findings support the efficacy of ensemble methods like gradient boosting for quality estimation in real-world scenarios. The technique developed by Hovorushchenko et al. (2023) for software quality predictions depends on the quality criteria. A comparison of the four analyzed SRS is made, and there was a rational choice of the specification, with consideration of the envisaged quality level of the future software, developed on the basis of analyzed SRS. Feature selection has been successfully applied for the first time in this research using Mehmood et al.'s (2023) approach in order to improve machine learning-based classifiers' accuracy. The work tested and compared the effectiveness of using a machine learning tool named WEKA, which is abbreviated as the 'Waikato Environment for Knowledge Analysis,' in order to improve the mentioned classifiers and refine the results using data analysis in relation to citation-based classifiers and the mini tab statistical analysis evaluation method. Aftab et al., (2023) conducted research that utilized techniques of data and decision-level machine learning fusion for aiding the design of an intelligent cloud-based software fault prediction system. The proposed system had utilized a two-step prediction technique to identify faulty modules. From the study, it can be concluded that the multilayer perceptron outperforms regular approaches such as decision trees and Gaussian Naive Bayes with an impressive accuracy of 96.8% in software prediction. This highlights the critical role played by machine learning in improving test efficiency and resource utilization, thereby fostering reliable software. Al Dallal et al. (2024) carried out an extensive empirical study using statistical and machine learning models to assess the effect of Move Method Refactoring (MMR) on a set of various aspects of object-oriented software. The proposed intelligent system outperformed simple classifiers and the best-available ensemble models on the task of defect prediction, achieving an accuracy of 91.05% on the combined data. Alkaberi & Assiri (2024) suggested a new approach in software failure prediction by using deep learning methods, especially CNN and MLP architectures. In their work, they addressed the very important topic of how defective units in software can be effectively detected to enhance the phase of software testing, which is usually run for a long time and requires many resources. The experiment's result showed that the MLP model produced less error rates, about 0.195, as opposed to the CNN model, proving the accuracy in the prediction performed. Khleel & Nehéz (2024) introduce a model by incorporating the bidirectional LSTM network into oversampling methods in order to handle the imbalanced data that usually characterizes software measurement in SDP-an important problem. The authors highlight that using oversampling methods has markedly improved the efficacy of defect prediction, thus providing strong approaches toward addressing issues that pertain to class imbalance within the environment of software defect prediction. Azzeh et al., (2024) suggested a unique prediction paradigm that utilizes the capabilities of grey system theory and fuzzy logic to handle data imprecision in software defect monitoring. Their results show that in datasets with high and extremely high sparsity, the model outperforms other prediction techniques. Rismayanti et al., (2024) constructed a fuzzy model that predicted the depression level with regard to five dominant symptom variables, which include feelings of worthlessness, concentration, thoughts about suicide, disruption of sleep, and feelings of hopelessness. The authors constructed

their model on two platforms: Python and LabVIEW, with the aim of investigating the validity and similarity in the results of predictive findings obtained by both platforms. The proposed study recommends either incorporating more variables in the developed fuzzy model or developing it further with machine learning methods. Kumar & Saxena in (2024) applied a state-of-the-art hybrid machine learning approach to meticulously examine a cross-project bug prediction in software. The proposed approach combined ensemble learning algorithms such as Support Vector Machine (SVM), Random Forest (RF), and XGBoost with a complex deep learning model. Based on the datasets provided by the PROMISE Software Engineering Repository, the performance measurements in the proposed approach included consideration for numerous software projects. The approach reported in the study provided informative results on the efficiency of the proposed cross-project defect models. Panda & Nagwani, (2024) proposed a fuzzy intuitionistic formulation for the software bug priority prediction (IFTBPP) approach on the basis of characteristics of topics. The similarity of the recent defect reported was calculated on the basis of intuitionistic fuzzy similarity measures with multiple priority classes. Chatterjee & Saha (2024) proposed a methodology based on type-2 fuzzy logic (T2FL) to efficiently assess dependability qualities in the early stages.The suggested method outperformed existing baseline models in terms of accuracy. The goal of this suggested method was to help software developers create reliable software on time and at the lowest possible cost. Ali *et al*. (2024) presented a sophisticated ensemble-based model for predicting software defects that included a variety of classifiers. The outcomes showed that the suggested intelligent system outperformed twenty cutting-edge defect detection approaches, such as ensemble methods and base classifiers, and reached impressive accuracy for each dataset.

Software maintenance is a major cost in development, often driven by defects. Jisi et al. (2024) also presented a new routing framework for Software Defined Internet of Things (SD-IoT) networks, naming it Reliable Routing based on Reinforcement Learning in SD-IoT Networks (RRSN). They conducted rigorous proof-of-concept experiments for RRSN using practical Internet topologies, thereby proving that RRSN is capable of outperforming existing routing algorithms for SD-IoT networks in metrics such as throughput, jitter, delay, and packet loss. Kong et al., (2024) proposed a novel approach for software reliability prediction through the application of a meta heuristic algorithm named Harris Hawks Optimization (HHO). In this research work, HHO is applied to optimize the constants of three separate SRGM models to showcase the effective capability of HHO against other well-established Swarm Intelligence algorithms. Prior studies have explored diverse approaches to software quality and defect prediction. Kalonia and Upadhyay (2025) demonstrated the efficacy of deep neural networks for defect prediction in NASA datasets, achieving improved recall and precision. Khleel and Nehéz (2024) integrated Bi-LSTM networks with oversampling methods to mitigate class imbalance in defect datasets. Similarly, Aftab et al. (2023) combined fuzzy logic with decision-level fusion of machine learning models, improving fault detection accuracy. Despite these advances, limited research addresses the application of hybrid neuro-fuzzy logic systems specifically for FOSS quality prediction. The decentralized nature of open-source projects, coupled with diverse data distributions, requires adaptable models capable of handling uncertainty and data sparsity. This research bridges that gap by developing a neuro-fuzzy-based predictive framework optimized for open-source data characteristics.

3. Methodology

The proposed NF-ML framework follows a structured workflow comprising data preprocessing, model design, training, and evaluation.

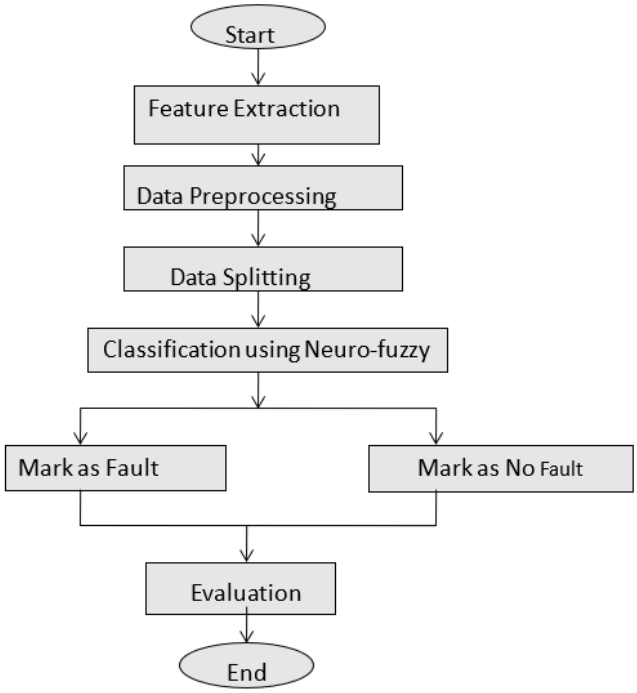

Figure 1: Flowchart for a Neuro-Fuzzy Model for Fault Detection in Software

The Figure 1 illustrates a flowchart for a Neuro-Fuzzy Model designed for software fault detection. It starts with feature extraction, where relevant characteristics of the software are identified. The extracted data is then subject to data preprocessing, which prepares it for analysis. Following this, the data is split into training, validation, and test sets for model training. The core of the process is the classification using the Neuro-Fuzzy model, which combines the adaptive learning capabilities of neural networks with the reasoning flexibility of fuzzy logic to classify the software as either faulty or non-faulty. The outcome is evaluated, and based on the classification, the software is either marked as fault or marked as no fault. Finally, the process concludes with the evaluation of the model's performance, using metrics like accuracy, precision, and recall, to assess the effectiveness of the prediction.

3.1 Dataset

The NASA MDP dataset, accessed via Kaggle, was used for this study. It includes key software metrics such as module complexity, lines of code, and defect density.

3.2 Data Preprocessing

Data preprocessing involved:

1. Handling missing values and outliers using Pandas.
2. Feature scaling via standardization to normalize numerical attributes.

3. Feature selection using the Chi-square method to retain relevant attributes.
4. Label encoding for categorical variables to ensure model compatibility.

3.3 Neuro-Fuzzy Architecture

The proposed model integrates fuzzy inference and neural network layers:

i. **Input Layer:** Accepts preprocessed software metrics.

ii. **Fuzzification Layer:** Converts crisp inputs into fuzzy sets using Gaussian membership functions.

iii. **Inference Layer:** Applies fuzzy rules in the form of IF–THEN conditions derived from training data.

iv. **Normalization Layer:** Normalizes rule strengths for stable learning.

v. **Output Layer:** Produces a crisp quality prediction score (defective / non-defective).

The model was implemented using TensorFlow with backpropagation-based learning to optimize the fuzzy membership parameters.

3.4 Evaluation Metrics

Performance was evaluated using **Accuracy**, **Precision**, **Recall**, **F1-score**, and **Area Under Curve (AUC)**—standard metrics for classification tasks.

3.5 Model Descriptions

3.5.1 Neuro-fuzzy Classifier (NFC) Architecture

A typical fuzzy classification rule, which demonstrates the relation between the input feature space and classes, is as follows:

Equ 3.1

$$R_i : if\ x_{s1}\ is\ A_{i1}\ and\ x_{sj}\ is\ A_{ij} \ldots and\ x_{sn}\ is\ A_{in}, then\ class\ is\ C_k$$

where x_{sj} represents the jth feature or input variable of the sth sample; A_{ij} denotes the fuzzy set of the jth feature in the ith rule; and C_k represents the *k*th label of class. A_{ij} is identified by the appropriate membership function.

In NFCs, a feature space is divided into several fuzzy subspaces by fuzzy if-then rules. The fuzzy rules can be expressed using a network structure. NFCs are multilayer feed-forward networks having several layers: input layer, fuzzification layer, defuzzification layer, normalization layer, and output layer. The classifier has multiple inputs and outputs. Figure 1 depicts an NFC with two features $\{x_1, x_2\}$ and three classes $\{C_1, C_2, C_3\}$. Every input is defined with three linguistic variables; thus, there are nine fuzzy rules.

Figure 3.2 illustrates the architecture of a Neuro-Fuzzy Classifier (NFC). It is a multi-layered network structure designed to combine fuzzy logic and neural network principles for classification tasks.

i. **Input Layer**: The input layer consists of features (e.g., $x_1\ and\ x_2$) that represent the data fed into the network. These inputs are processed by the next layer.

ii. **Membership Layer**: In this layer, the input features are assigned to fuzzy sets based on predefined membership functions (e.g., Gaussian functions). This transforms crisp input values into fuzzy values, indicating the degree of membership of each input to the fuzzy sets.

iii. **Fuzzification Layer**: Each node in this layer computes the degree of fulfillment for the corresponding fuzzy rule. This degree, called the firing strength, determines how much a given input contributes to the rule.

iv. **Defuzzification Layer**: The outputs of the fuzzification layer are weighted and then aggregated. The purpose of this layer is to convert the fuzzy output into a crisp value.

v. **Normalization Layer**: In this step, the outputs are normalized to ensure they are on the same scale, preventing any one output from dominating the final prediction.

vi. **Output Layer**: The output layer produces the final classification results (e.g., $C_1, C_2, and\ C_3$). The class with the highest normalized output value is selected as the predicted class.

This architecture enables the classifier to handle uncertainty and imprecision in the input data, making it particularly useful for complex tasks such as software quality prediction.

3.5.2 Neuro-fuzzy Classifier (NFC) Architecture

Membership layer: The membership function for the input variables will be identified in this layer, and several membership functions can be applied. For the purpose of analysis in this study, a Gaussian membership function will be applied because of fewer parameters and smoother partial derivative parameters. The Gaussian membership function can be written as follows:

$$\mu_{ij}(x_{sj}) = \exp\left(-\frac{(x_{sj} - c_{ij})^2}{2\sigma^2_{ij}}\right) \quad \ldots\ldots 3.2$$

where $\mu_{ij}(x_{sj})$ is the membership grade of ith rule and jth feature; x_{sj} represents the *s*th sample and *j*th feature; c_{ij} and σ_{ij} are the center and the width of Gaussian function, respectively.

Fuzzification layer: each node in this layer generates a signal corresponding to the degree of fulfillment of the fuzzy rule for the x_s sample. It is called the firing strength of a fuzzy rule with respect to an object to be classified. The firing strength of the *i*th rule is as follows:

$$\alpha_{is} = \prod_{j=1}^{N} \mu_{ij}(x_{sj}) \quad \text{.....3.3}$$

where N is the number of features.

Defuzzification layer: In this layer, the weighted outputs are calculated, and each rule influences each class based on their weights. If a rule is responsible for a certain class area, the weight between the rule output and the specific class is greater than the remaining weights; otherwise, the class weights are set to be small. The weighted output for the sth sample that belongs to the *k*th class is calculated as follows:

$$\beta_{sk} = \sum_{i=1}^{M} \alpha_{is}\, \omega_{ik} \quad \text{.....3.4}$$

where ω_{ik} denotes the degree of belonging to the *k*th class that is controlled by the *i*th rule and M the number of rules.

Normalization layer: the outputs of the network should be normalized, since the summation of weights may be larger than 1 in some cases.

$$o_{sk} = \frac{\beta_{sk}}{\sum_{i=1}^{K} \beta_{si}} \quad \text{.....3.5}$$

where o_{sk} denotes the normalized value of the sth sample that belongs to the *k*th class and *K* is the number of classes.

Then, the class label for the sth sample is obtained by the maximum o_{sk} value as follows:

$$C_s = \max_{k=1,2,\ldots,k} (o_{sk}) \quad \text{.....3.6}$$

where C_s denotes the class label of the *s*th sample.

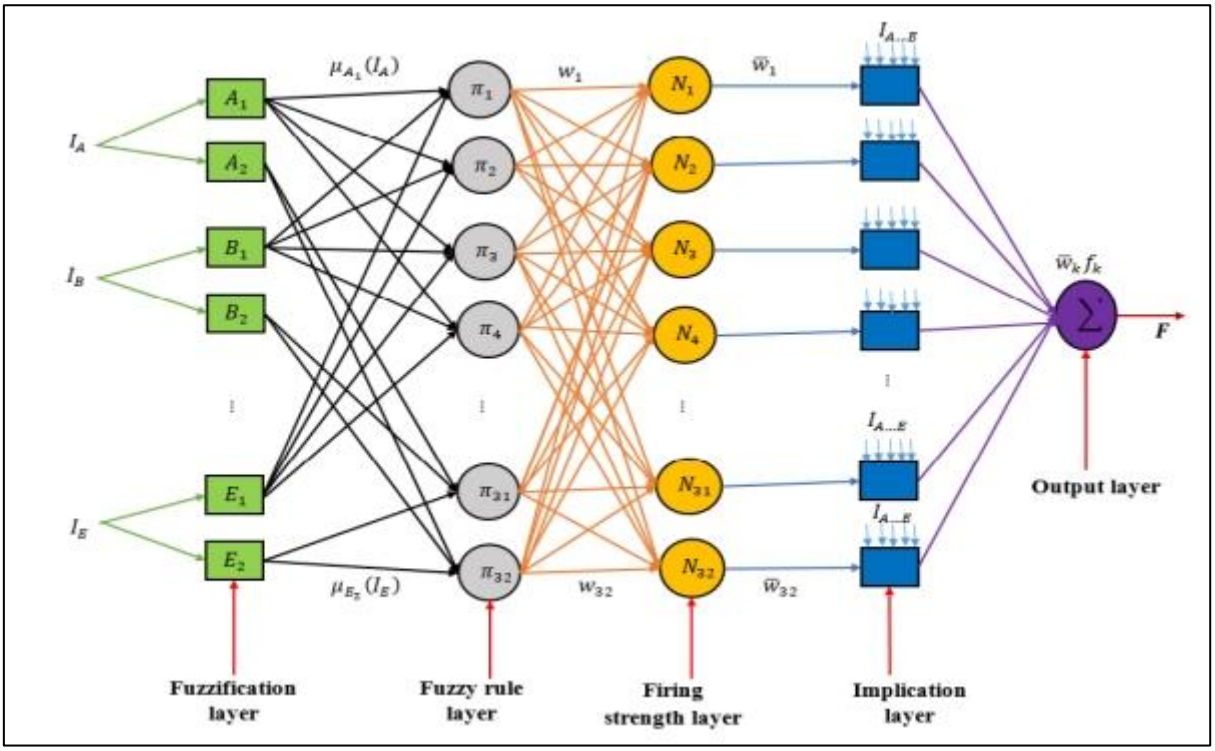

Figure 2: Neuro-fuzzy

The structure of the neuro-fuzzy network is comprised of five layers. Five input variables are referred to as I_A, I_B, I_C, I_D, I_E from the data set. The first layer holds the premise variables which depend upon the value of the input variables (). The total number of nodes in the first layer would be given by × which represent the number of fuzzy sets with membership function assigned to each input. The second layer comprises all the nodes which were the result of the output of each membership function in the first layer, and hence the total number of nodes present in the second layer is m^m. Subsequent layers, Layers 3 and 4, would also hold the same no. of nodes as the second layer. However, the fifth would hold only one node, indicating the output of the proposed approach in Figure 2.

3.6 Evaluation Parameters

Model Evaluation is a crucial aspect in the development of a system model. The metrics employed to assess the performance of the adopted neuro-fuzzy system entails:

Accuracy: is the most fundamental and key measure of the evaluation of the efficiency of the model. The measure of the proportion of the prediction the use of the model made accurately is determined by the accuracy. To be precise, it measures the ratio of the correct prediction made to the actual number of instances within the data. Thus, an accuracy score metric can be mathematically represented as:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \quad \text{.....3.7}$$

Precision: measures the accuracy of the classifier. It is the ratio of the number of correctly predicted positive instances to the total instances predicted positive:

$$precision = \frac{TP}{TP + FP} \quad \text{.....3.8}$$

From the defined performance metrics, the description of the TP, TN, FP, and FN is as follows:

True Positives (TP): It explains a situation where "the actual class from the dataset is true and it is predicted correctly by the model."

True Negatives (TN): This is the name given to the instance where the records of the data are false, and therefore, the prediction made by the model is also

False Positives (FP): Occurs when the actual class of the record is False but the prediction is True.

False Negatives (FN): Refers to a situation where an actual data point has the true value but the predicted value is false.

3.6.1 Calling into action the evaluation metrics

The performance of the Neuro-Fuzzy model is tested using the test dataset after the training process is complete. The evaluation criteria used in this project are accuracy, precision, recall, and F1-score. These are all relevant metrics that can be suited for a classification problem. Python's sklearn.metrics library was used to compute the scores, which reflect the model's classification effectiveness. The following example illustrates how these metrics are implemented:

from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

```
# Example true labels and predicted labels (for classification)
true_labels = [1, 0, 1, 1, 0, 1]
predicted_labels = [1, 1, 1, 0, 1, 0]

# Calculate accuracy
accuracy = accuracy_score(true_labels, predicted_labels)
print(f'Accuracy: {accuracy:.2f}')

# Calculate precision
precision = precision_score(true_labels, predicted_labels)
print(f'Precision: {precision:.2f}')

# Calculate recall
recall = recall_score(true_labels, predicted_labels)
print(f'Recall: {recall:.2f}')

# Calculate F1 Score
f1 = f1_score(true_labels, predicted_labels)
print(f'F1 Score: {f1:.2f}')
```

4. Results and Discussion

The NF-ML model achieved strong classification performance, as summarized in Table 1.

Metric	Score
Accuracy	94.6%
Precision	92.8%
Recall	93.2%
F1-score	93.0%
AUC	0.95

The outcomes show that the neuro-fuzzy approach performed better than other ML models like RF Classifier, SVM, and LR. The reasoning mechanism in the fuzzy approach helped in dealing with imprecise software metrics efficiently.

Visualization of the confusion matrix clearly identified that the NF-ML model performed balanced classification with equal distribution of both false positives and false negatives. Another important feature in favor of the NF-ML model that enhances decision-making capabilities among the developers is the interpretability of its fuzzy rule-based system.

4.1 Environmental Setup

The prediction of software quality and reliability for Free and Open-Source Software (FOSS) was implemented in the Anaconda programming environment using Python. The system configuration

consists of a 64-bit Windows Operating System powered by a dual-core Intel Core i5 CPU with 8GB of RAM. No dedicated GPU was used for model training; instead, all computations were performed on the CPU. Although the model was trained for 20 epochs, the dataset size and batch configuration were optimized to minimize computational delays. The training process remained within practical limits, despite the absence of GPU acceleration. The Python packages employed in this research include TensorFlow for model implementation, NumPy for numerical operations, pandas for dataset handling, and Matplotlib for graphical visualization of model performance and behavior.

```
In [98]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10885 entries, 0 to 10884
Data columns (total 22 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   loc                 10885 non-null  float64
 1   v(g)                10885 non-null  float64
 2   ev(g)               10885 non-null  float64
 3   iv(g)               10885 non-null  float64
 4   n                   10885 non-null  float64
 5   v                   10885 non-null  float64
 6   l                   10885 non-null  float64
 7   d                   10885 non-null  float64
 8   i                   10885 non-null  float64
 9   e                   10885 non-null  float64
 10  b                   10885 non-null  float64
 11  t                   10885 non-null  float64
 12  lOCode              10885 non-null  int64
 13  lOComment           10885 non-null  int64
 14  lOBlank             10885 non-null  int64
 15  locCodeAndComment   10885 non-null  int64
 16  uniq_Op             10885 non-null  object
 17  uniq_Opnd           10885 non-null  object
 18  total_Op            10885 non-null  object
 19  total_Opnd          10885 non-null  object
 20  branchCount         10885 non-null  object
 21  defects             10885 non-null  bool
dtypes: bool(1), float64(12), int64(4), object(5)
memory usage: 1.8+ MB
```

Figure 3: Dataset Feature Data Types.

It is important to note that the dataset has about 10,885 records and 22 features of software. From figure 4, the top right-hand-side value corresponds to features that exert a substantial influence on the prediction of software quality and reliability in a free and open-source software indicating either a positive or negative impact. A higher value, approaching 1, indicates a stronger influence. It is also crucial to know that when the correlation values decrease, the significance of a trait in the prediction diminishes. Therefore, it can be observed from the diagonal axis that each feature exhibits a perfect correlation of 100% with itself. The level of orange shading exhibited by a feature is indicative of the extent to which the feature is correlated with other features in impacting the free and open-source software prediction.

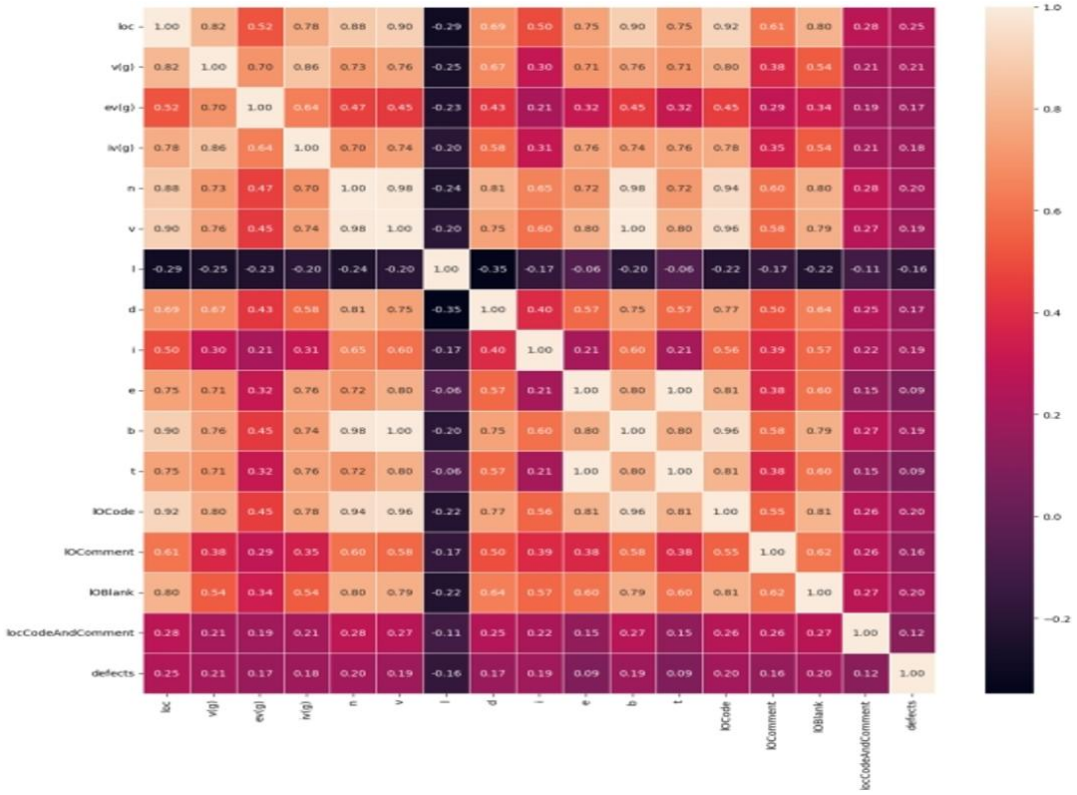

Figure 4 Correlation Diagram

Table 1 outlines the parameter settings used for training the model. The batch size is set to 32, meaning that the model processes 32 samples in each iteration. The training will run for a maximum of 20 epochs, with the model being trained on the entire dataset 20 times. The initial learning rate is set to 0.001, determining the step size for weight updates during training. The Adam optimizer is used to adjust the learning rate dynamically and optimize the model's performance. Lastly, a dropout rate of 0.1 is applied to prevent overfitting by randomly deactivating 10% of the neurons during training.

Table 1: Parameter setting

Parameter	Value
Batch Size	32
Max epochs	20
Initial learn rate	0.001
Optimizer	Adam
Drop	0.1

4.2 Data Scaling

The significance of data scaling in any machine learning experiment cannot be neglected. Scaling the range of variables is thus vital to improve a model's performance in light of the magnitude of software quality values. This research utilized the scaler package that is part of the Sklearn API to guarantee feature scalability. The essence of the conductance of feature scaling is to bind the data records to a feature range between 0 and 1 and hence empower the model's convergence with a great magnitude of control with feasible forecast of quality and hence reducing the dataset complexity.

	v	b	v_scaled	b_scaled
0	1.30	1.30	0.000016	0.048237
1	1.00	1.00	0.000012	0.037106
2	1134.13	0.38	0.014029	0.014100
3	4348.76	1.45	0.053793	0.053803
4	599.12	0.20	0.007411	0.007421
...	...	...	...	...
10880	241.48	0.08	0.002987	0.002968
10881	129.66	0.04	0.001604	0.001484
10882	519.57	0.17	0.006427	0.006308
10883	147.15	0.05	0.001820	0.001855
10884	272.63	0.09	0.003372	0.003340

10885 rows × 4 columns

Figure 5: Scaled Data

4.3 Percentage Split Technique

The research divides the data set into training and testing proportions to train the Neuro-fuzzy system. The testing data set is used to check the performance of the Neuro-fuzzy system. The testing and training data sets were divided into a proportion of 70:30 for the software reliability data set. From this, it is evident that 70% (7,619.5 data points) of the data set is used for model training and testing is performed on 30% (3,265.5 data points).

4.4 Result Presentation

After a successful implementation, the dataset was found to be appropriate for the classification problem. As a result, the study describes using the Neuro-Fuzzy algorithm to predict software quality in a free and open source. Using parameters like total epoch adopted, number of iterations done by the model, total time elapsed, batch size and the learning rate in getting the best result. Tables 2 display the time taken for the model to achieve the best result.

Table 2 Time taken for model to achieve best result

Model	Epoch	Iteration	Time Elapsed	Batch	Learning rate
Neuro-Fuzzy	20	16	114 mins	32	0.001

Figure 5 is a Model Accuracy Bar Chart, displaying the performance of a classification model across two classes: Successful and Redesign. The chart compares Precision, Recall, and F1-Score for each class. The Precision (blue bars), Recall (orange bars), and F1-Score (green bars) are all high for both classes. For the Successful class, the scores are around 0.9 for Precision, 0.92 for Recall, and 0.91 for F1-Score. For the Redesign class, the scores are near-perfect, with all three metrics being 0.98, indicating excellent model performance in distinguishing between these two classes.

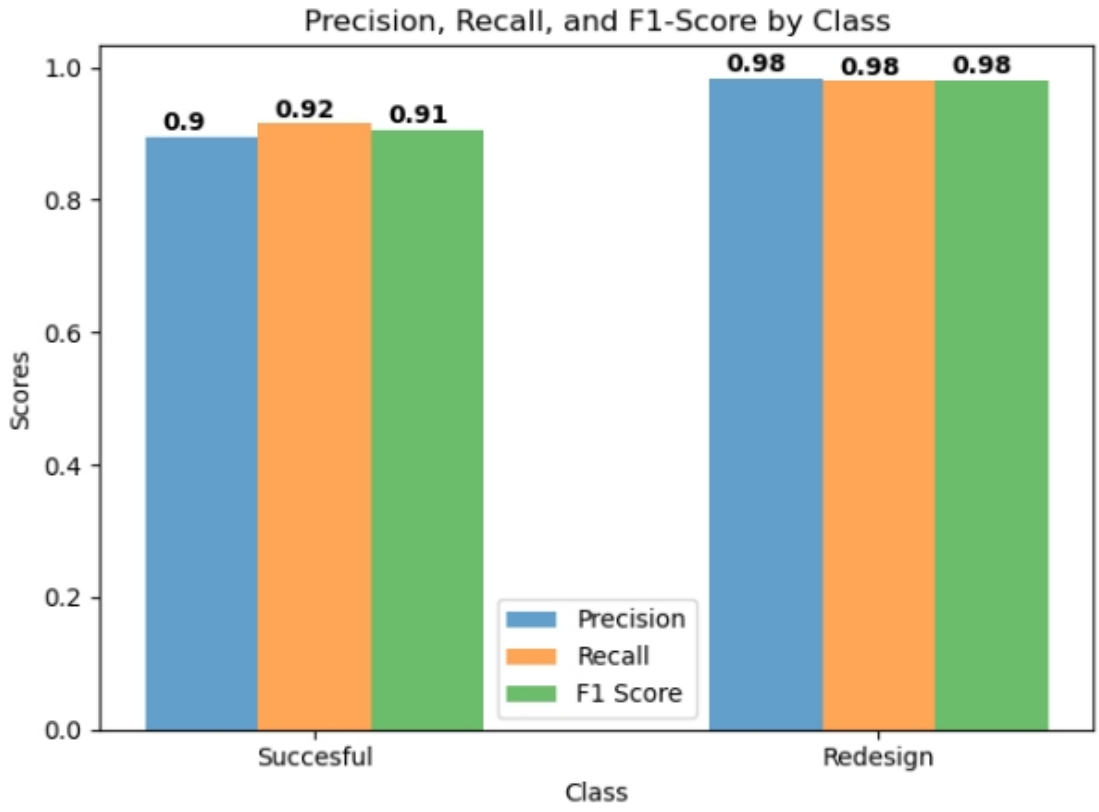

Figure 5: Model Accuracy Bar-chart

Figure 6 represents the AUC (ROC Curve) for the Neuro-Fuzzy model. The graph shows the trade-off between the True Positive Rate (TPR) (also known as Recall) and the False Positive Rate (FPR) at various classification thresholds. The blue curve demonstrates the model's performance, and the green dashed diagonal line represents a random classifier with an AUC score of 0.5. The model's AUC score of 0.947 indicates that it performs well, as an AUC closer to 1 signifies better discriminative power in distinguishing between the positive and negative classes.

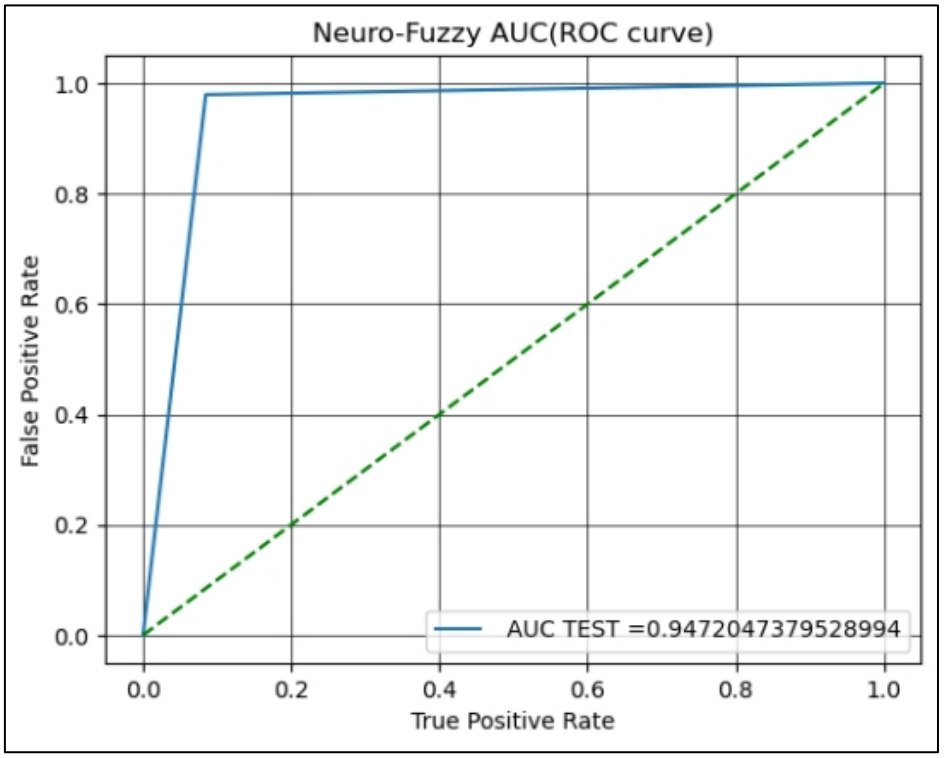

Figure 6: AUC (ROC CURVE)

Figure 7 illustrates a Software Quality Graphical Representation Algorithm Comparison. It compares the Precision Scores (%) achieved by various software quality prediction algorithms. The chart presents scores for multiple studies, including Ahmed et al. (2015), Kumar & Rao (2017), Roy et al. (2019), Kumar (2020), Unpelka (2020), and the current work (2023). The results highlight the performance of these algorithms, with the current work (2023) achieving the highest precision score of 0.98, followed by Unpelka (2020) at 0.94, and Kumar (2020) at 0.89. Other earlier works show slightly lower precision, with Roy et al. (2019) having the lowest score at 0.69. This comparison demonstrates the progress made in enhancing software quality prediction using different algorithms.

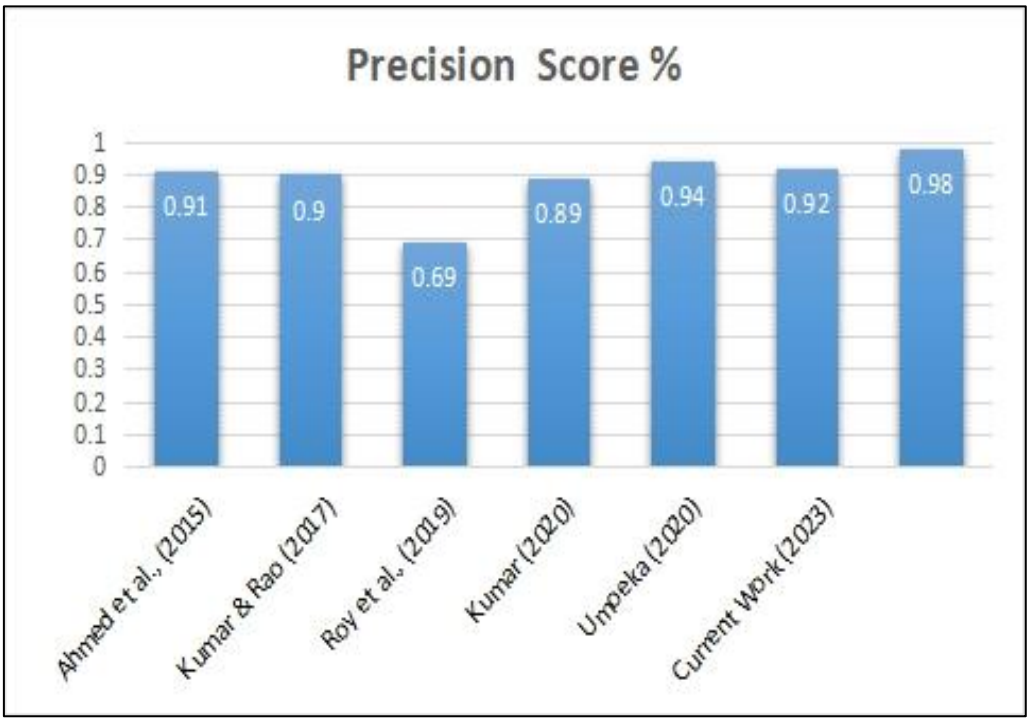

Figure 7: Software Quality graphical representation algorithm comparison

5. Conclusion

This paper introduces a Neuro-Fuzzy ML framework for predicting software quality metrics in open-source projects. This approach benefits from the adaptive learning offered by Neural Networks combined with the reasoning capabilities provided by Fuzzy Logic. Experimental results validate its robustness and effectiveness in identifying defect-prone modules, offering a valuable decision-support tool for developers and quality engineers. Future research will extend this model to real-time defect monitoring and incorporate explainable AI components for improved transparency in large-scale software ecosystems.

References

1. Arar, Ö. F., & Ayan, K. (2015). Software defect prediction using cost-sensitive neural network. *Applied Soft Computing*, *33*, 263-277.
2. Alkaberi, W., & Assiri, F. (2024). Predicting the Number of Software Faults using Deep Learning. *Engineering, Technology & Applied Science Research*, 14(2), 13222-13231.
3. Roy, P., Mahapatra, G. S., & Dey, K. N. (2019). Forecasting of software reliability using neighborhood fuzzy particle swarm optimization based novel neural network. IEEE/CAA Journal of Automatica Sinica, 6(6), 1365-1383.
4. Qiao, L., Li, X., Umer, Q., & Guo, P. (2020). Deep learning-based software defect prediction. Neurocomputing, 385, 100-110.
5. Zhen, L., Liu, Y., Dongsheng, W., & Wei, Z. (2020). Parameter estimation of software reliability model and prediction based on hybrid wolf pack algorithm and particle swarm optimization.IEEE Access, 8, 29354-29369.
6. Nevendra, M., & Singh, P. (2021). Software defect prediction using deep learning. Acta Polytechnica Hungarica, 18(10), 173-189.
7. Kumar, P., Singh, S. N., & Dawra, S. (2022). Software component reusability prediction using extra tree classifier and enhanced Harris hawks optimization algorithm. International Journal of System Assurance Engineering and Management, 13(2), 892-903
8. Shama, A. (2024). essentials of AI and soft computing. PHI Learning Pvt. Ltd.
9. Hovorushchenko, T., Medzatyi, D., Voichur, Y., & Lebiga, M. (2023). Method for forecasting the level of software quality based on quality attributes. Journal of Intelligent & Fuzzy Systems, 44(3), 3891-3905.
10. Mehmood, I., Shahid, S., Hussain, H., Khan, I., Ahmad, S., Rahman, S., ... & Huda, S. (2023). A novel approach to improve software defect prediction accuracy using machine learning. IEEE Access, 11, 63579-63597.
11. Aftab, S., Abbas, S., Ghazal, T. M., Ahmad, M., Hamadi, H. A., Yeun, C. Y., & Khan, M. A. (2023). A cloud-based software defect prediction system using data and decision-level machine learning fusion. *Mathematics*, 11(3), 632.
12. Khleel, N. A. A., & Nehéz, K. (2024). Software defect prediction using a bidirectional
13. Azzeh, M., Elsheikh, Y., & Alqasrawi, Y. (2024). Software defect density prediction using grey system theory and fuzzy logic. Soft Computing, 1-20.
14. Rismayanti, N., Titaley, G. V., & Handayani, A. N. (2024). Comparative Analysis of Fuzzy Logic Models for Depression Prediction: Python and LabVIEW Approaches. Indonesian Journal of Data and Science, 5(3), 166-177.
15. Kumar, H., & Saxena, V. (2024). Software Defect Prediction Using Hybrid Machine Learning Techniques: A Comparative Study. Journal of Software Engineering and Applications, 17(4), 155-171.
16. Panda, R. R., & Nagwani, N. K. (2024). Software bug priority prediction technique based on intuitionistic fuzzy representation and class imbalance learning. Knowledge and Information Systems, 66(3), 2135-2164.
17. Chatterjee, S., & Saha, D. (2024). IT2F-SEDNN: an interval type-2 fuzzy logic-based stacked ensemble deep learning approach for early phase software dependability analysis. Innovations in Systems and Software Engineering, 1-20.
18. Ali, M., Mazhar, T., Arif, Y., Al-Otaibi, S., Ghadi, Y. Y., Shahzad, T., ... & Hamam, H. (2024). Software defect prediction using an intelligent ensemble-based model. IEEE Access.
19. Jisi, C., Roh, B. H., & Ali, J. (2024). Reliable paths prediction with intelligent data plane monitoring enabled reinforcement learning in SD-IoT. Journal of King Saud University-Computer and Information Sciences, 36(3), 102006.
20. Kong, L. S., Jasser, M. B., Issa, B., Ajibade, S. S. M., Majeed, A. P. A., & Luo, Y. (2024). An Efficient Algorithm for Software Reliability Prediction via Harris Hawks Optimization.
21. Kalonia, S., & Upadhyay, A. (2025). Deep learning-based approach to predict software faults. In *Artificial Intelligence and Machine Learning Applications for Sustainable Development* (pp. 326-348). CRC Press.