AI-DRIVEN APPROACH FOR SOFTWARE QUALITY PREDICTION TO IMPROVE ESTIMATION ACCURACY

Dr. M. Swapna[1*], E. Rishitha[2], Vikas Reddy[2], Yedla Yeshwanth[2], J. Abhiraj[2]

[1]Assistant Professor, [2]UG Student, [1,2]Department of Computer Science and Engineering

Vaagdevi College of Engineering(UGC - Autonomus), Bollikunta, Warangal, Telangana.

*Corresponding author: Dr. M. Swapna(swapna_m@vaagdevi.edu.in)

ABSTRACT

Software quality prediction is crucial in enhancing estimation accuracy and reducing post-release defects in software engineering. Studies reveal that over 60% of software projects exceed budget or time estimates, while 45% of software defects remain undetected until after deployment, and nearly 30% of testing efforts are spent on low-risk modules. Despite these concerns, manual software quality estimation remains error-prone due to subjectivity, lack of real-time analytical support, and inefficient handling of high-dimensional data. This research proposes an automated software quality prediction model using a labeled dataset with six quality categories: ('Failure', 'Low', 'Low Medium', 'Medium High', 'High', 'Very High'). The dataset undergoes rigorous preprocessing including null value treatment, label encoding, normalization, and outlier detection. To address dimensionality, Principal Component Analysis (PCA) is employed for optimal feature extraction and variance preservation. Comparative analysis is conducted using existing classifiers—Bernoulli Naive Bayes (NBC), Decision Tree Classifier (DTC), Random Forest Classifier (RFC), Logistic Regression Classifier (LRC), and Bagging Ensemble Method—highlighting their limitations in handling multi-class predictions and overfitting issues. A Gradient Boosting Classifier is proposed to enhance classification precision and reduce bias-variance trade-off by sequentially improving weak learners. Experimental results confirm that the proposed model significantly improves estimation accuracy across all six categories, outperforming traditional classifiers in terms of F1-score, precision, and recall.

Keywords: Software quality prediction, Quality categories, Principal Component Analysis (PCA), Gradient Boosting Classifier, Automated prediction model.

1. INTRODUCTION

In today's software-centric world, ensuring quality and reliability of software products has become increasingly crucial. As the demand for high-performance and bug-free applications continues to rise, the consequences of delivering low-quality software can be significant. According to the Consortium for IT Software Quality (CISQ), poor-quality software cost the United States economy over $2.08 trillion in 2020 alone. Moreover, software failures in critical systems such as healthcare, banking, and aviation can result in not only financial loss but also pose risks to human life and safety.A substantial portion of development resources, often as much as 40%, is dedicated to testing and quality assurance. However, despite this investment, many issues still escape into production due to ineffective quality estimation during the early stages. The Standish Group's CHAOS Report indicates that approximately 52% of software projects suffer from underperformance or outright failure due to quality issues. These findings emphasize the growing need for proactive quality management strategies throughout the software development lifecycle.

2. LITERATURE SURVEY

Chowdhury, Rajarshi Roy, et al. [1] proposed device fingerprinting model demonstrates over 99% and 95% precisions in distinguishing between known and unknown traffic traces and in identifying IoT and non-IoT traffic traces, respectively. 98.49% precision has also been demonstrated on an individual device classification task. These results are significant as the model can be utilized to effectively secure a resource-constrained IoT network, which despite its rapid growth of usage, is more prone to attack, partly due to its dependence on traditional explicit identification methods. Kotak, Jaidip , et al. [2] proposed approach is applicable for any IoT device, regardless of the protocol used for communication. As our approach relies on the network communication payload, it is also applicable for the IoT devices behind a network address translation (NAT) enabled router. In this study, we trained various classifiers on a publicly accessible dataset to identify IoT devices in different scenarios, including the identification of known and unknown IoT devices, achieving over 99% overall average detection accuracy.

Carson, et al. [3] proposed a Machine-learning-assisted approaches are promising for device identification since they can capture dynamic device behaviors and have automation capabilities. Supervised machine-learning-assisted techniques demonstrate high accuracies for device identification. However, they require a large number of labeled datasets, which can be a challenge. On the other hand, unsupervised machine learning can also reach good accuracies without requiring labeled datasets. This paper presents an unsupervised machine-learning approach for IoT device identification. Elngar, Ahmed, et al. [4] paper implements a methodology for network traffic classification using clustering, feature extraction, and variety for the Internet of Things (IoT). Further, K-Means is used for network traffic clustering datasets, and feature extraction is performed on grouped information. KNN, Naïve Bayes, and Decision Tree classification methods classify network traffic because of extracted features, which presents a performance measurement between these classification algorithms. The results discuss the best machine learning algorithm for network congestion classification.

Zarzoor, Ahmed et al. [5] proposed model performance is evaluated according to evaluation metrics: accuracy, precision, recall and F1-score and energy usage in comparison with two models: ML based Support Vector Machine IoT-TCSVM and ML based Deep Neural Network (IoT-TCDNN). The evaluations result has been shown that IoT TCSNN consumes less energy in contrast to IoT-TCDNN and IoT-TCSVM. Also, it gives high accuracy in comparison with IoT-TCSVM Elhaloui, et al. [6] proposed model performance is evaluated according to evaluation metrics: accuracy, precision, recall and F1-score and energy usage in comparison with two models: ML based Support Vector Machine IoT-TCSVM and ML based Deep Neural Network (IoT-TCDNN). The evaluations result has been shown that IoT TCSNN consumes less energy in contrast to IoT-TCDNN and IoT-TCSVM

Malathi, et al. [7] proposed goal of this article ability to understand the efficiency of machine learning (ML) algorithms in opposing Network-related cyber security Assault, with a focus on Denial of Service (DoS) attacks. We also address the difficulties that require to be discussed to implement these Machine Learning (ML) security schemes in practical physical object (IoT) systems. In this research, our main aim is to provide security by multiple machine-learning (ML) algorithms that are mostly used to recognise the interrelated (IoT) network Assault immediately. Unique metadata, BotIoT, is accustomed to estimate different recognition algorithms. Senthil Kumaran, et al. [8] suggested approach minimizes the amount of time spent on optimization operations while maintaining the predictive performance of the induced uncertain models. We work out the entropy standards of traffic characteristics and categorize the uncertain traffic using ML techniques. Our technique successfully identifies devices in a variety of uncertain network situations, with consistent performance in all

scenarios. Our technique is also resistant to unpredictability in network behavior, with abnormalities or uncertainties propagating throughout the network. Belkadi, et al. [9] proposed and showed that the supervised algorithms used (Naive Bayes, SVM (SMO), Random Forest, C4.5 (J48)) gave promising results of up to 97% when using the studied features and over 95% when using the generated features.

3. PROPOSED SYSTEM

Step 1: Dataset Collection and Labeling: The first step involves gathering a comprehensive software quality dataset with labeled categories: *['Failure', 'Low', 'Low Medium', 'Medium High', 'High', 'Very High']*. The dataset may include various features such as code metrics, defect data, and developer experience, among others.

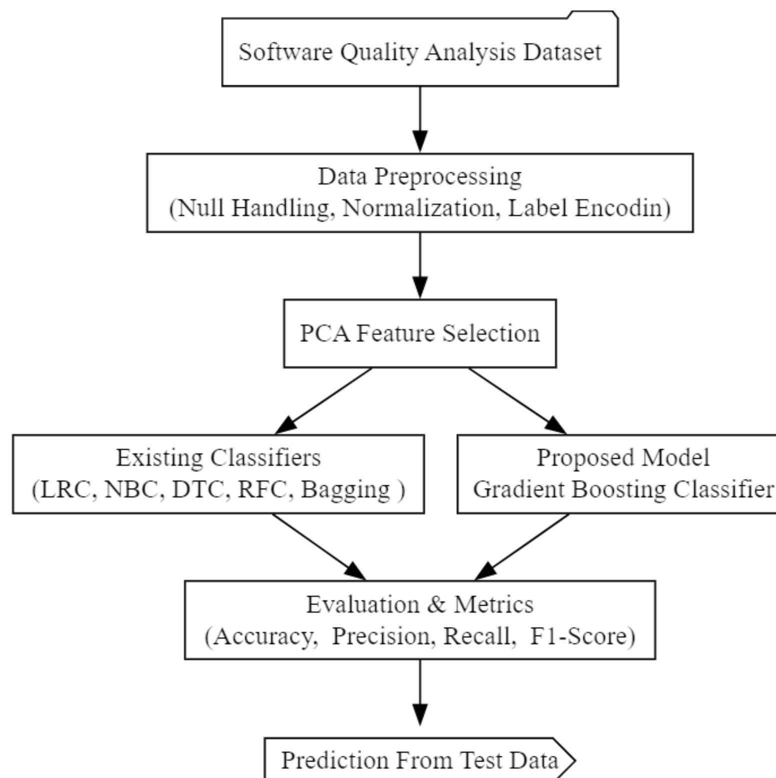

Fig. 1: Block Diagram Proposed System architecture.

Step 2: Data Preprocessing: Any missing or incomplete data is treated through imputation or deletion based on the data distribution. Data is scaled using techniques like Min-Max scaling to bring all features to a consistent range for better model convergence. Categorical variables are transformed into numerical format using label encoding to make them usable for machine learning models.

Step 3: Feature Selection with PCA: Principal Component Analysis (PCA) is employed to reduce the feature set dimensionality while retaining the most significant variance within the data. This ensures that the model can handle high-dimensional data efficiently without overfitting or losing important information.

Step 4: Model Selection and Training: Existing classifiers such as Bernoulli Naive Bayes (NBC), Decision Tree Classifier (DTC), Random Forest Classifier (RFC), Logistic Regression Classifier (LRC), and Bagging are tested on the preprocessed dataset to compare baseline performances. Gradient Boosting Classifier is selected as the primary model due to its ability to reduce bias and variance, sequentially improving weak learners, which outperforms traditional models. A weighted

ensemble learning method is applied, combining predictions from DTC, RFC, LRC, and Bagging classifiers, and assigning different weights based on their individual performance metrics. This allows the model to benefit from each classifier's strengths and minimize its weaknesses.

Step 5: Evaluation and Results Analysis: The model's performance is evaluated using accuracy, precision, recall, F1-score, and confusion matrix to compare the prediction accuracy across different categories. The Gradient Boosting Classifier with ensemble learning and PCA feature selection is shown to provide superior results compared to existing classifiers, with improved classification accuracy across all six quality categories.

3.2 Data Preprocessing

Step 1: Handling Missing Values: The dataset is initially checked for missing or null entries across all columns. If any null values are present, they are replaced with zero. This step ensures that there are no NaN (Not a Number) entries that could disrupt the training or processing of machine learning algorithms. Filling with zero is a simple yet effective strategy when the missing data is minimal or when zero is a neutral substitute that doesn't bias the dataset significantly.

Step 2: Label Encoding Categorical Columns Several columns in the dataset, such as 'QualifiedName', 'Name', 'Complexity', 'Coupling', 'Size', and 'Lack of Cohesion', are categorical or contain string values. These need to be converted into numerical form before being used by machine learning models. This is achieved using a label encoding technique where each unique string value in a column is mapped to a unique integer. The transformation ensures that all these attributes become numeric and suitable for further analysis.

Step 3: Separating the Output Label From the label-encoded dataset, the column 'Complexity' is extracted and stored as the target variable or output label. This column represents the software quality category or class that the model will eventually be trained to predict. It is separated from the dataset to prevent it from being treated as an input feature during model training.

Step 4: Dropping the Target Feature from Input Set Once the output label is extracted, the 'Complexity' column is removed from the main dataset. This step ensures a clear distinction between the input features and the output variable, which is essential in supervised learning tasks to avoid data leakage and ensure model integrity.

Step 5: Normalizing the Dataset After encoding and separating the target column, the remaining features are subjected to normalization. Normalization transforms the values of features to a common scale, typically in the range of 0 to 1. This step is crucial for algorithms that are sensitive to the scale of input features, such as gradient-based optimizers, ensuring that no single feature dominates the training process due to its magnitude.

Step 6: Outputting Data Preview and Category Info Finally, the processed dataset and the list of predefined categories such as ['Failure','Low','Low Medium','Medium High','High','Very High'] are displayed in the output interface. This step is intended for user feedback and confirmation, helping verify that the preprocessing steps have been applied correctly and allowing a visual check of the dataset's current structure.

3.3 PCA Feature Selection

Principal Component Analysis (PCA) offers a powerful way to reduce the dimensionality of a dataset while preserving as much meaningful variance as possible. One of the key benefits is that it transforms the original features into new uncorrelated components, which simplifies the structure of the data and helps improve the performance of classification models. In the context of software

quality prediction, where features like complexity, coupling, and cohesion may be interrelated or noisy, PCA filters out redundancy and focuses on the most informative aspects of the data. However, the input features should be application-specific because PCA is data-driven and doesn't consider feature semantics. For instance, in software engineering datasets, choosing features that genuinely represent software metrics is important before applying PCA, as the effectiveness of PCA depends on the relevance of initial features.

Step 1: Standardization of Features The first operation in PCA involves standardizing the dataset. This is done to ensure that each feature contributes equally to the analysis. Since PCA is sensitive to the scale of the data, features with larger numerical ranges can dominate the results. Standardization shifts the data to have a mean of zero and scales it to have unit variance.

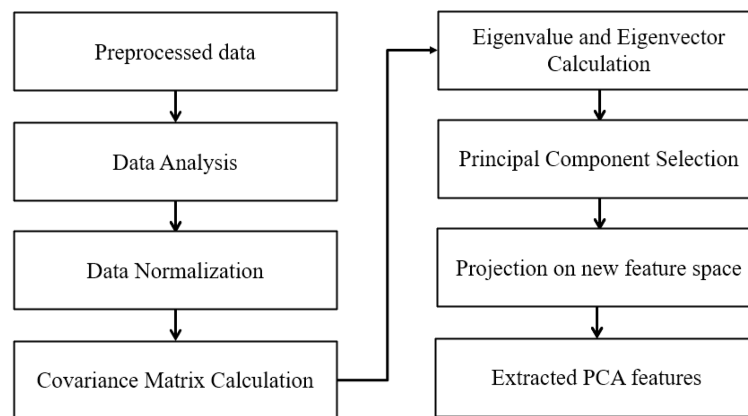

Fig. 2: PCA Block Diagram.

Step 2: Covariance Matrix Computation Once the data is standardized, the next step is to compute the covariance matrix. This matrix captures how features vary with respect to one another. If features are strongly correlated, the covariance values will be high. This matrix helps identify the directions (principal components) in which the data varies the most.

Step 3: Eigenvalue and Eigenvector Calculation From the covariance matrix, PCA calculates eigenvalues and corresponding eigenvectors. Eigenvectors determine the direction of the new feature space, while eigenvalues determine the magnitude or importance of those directions. The eigenvectors with the highest eigenvalues are considered the most significant principal components.

Step 4: Selection of Principal Components After computing eigenvectors and eigenvalues, the top k eigenvectors (based on eigenvalue magnitude) are selected. These form the new reduced feature space. The number of principal components chosen depends on how much of the total variance the user wants to retain—usually determined by a threshold like 95% of the cumulative variance.

Step 5: Projection onto the New Feature Space Finally, the original data is projected onto the newly selected principal components. This results in a new dataset with reduced dimensions, where each record is represented in terms of the principal components rather than the original features. This transformed dataset retains the essential patterns while reducing noise and redundancy.

3.4 Model Building

3.4.1 Proposed Gradient Boosting Classifier

The Gradient Boosting Classifier (GBC) operational diagram presented in Figure 4.7, which effectively predict the software quality category, providing valuable insights for software development and maintenance processes.

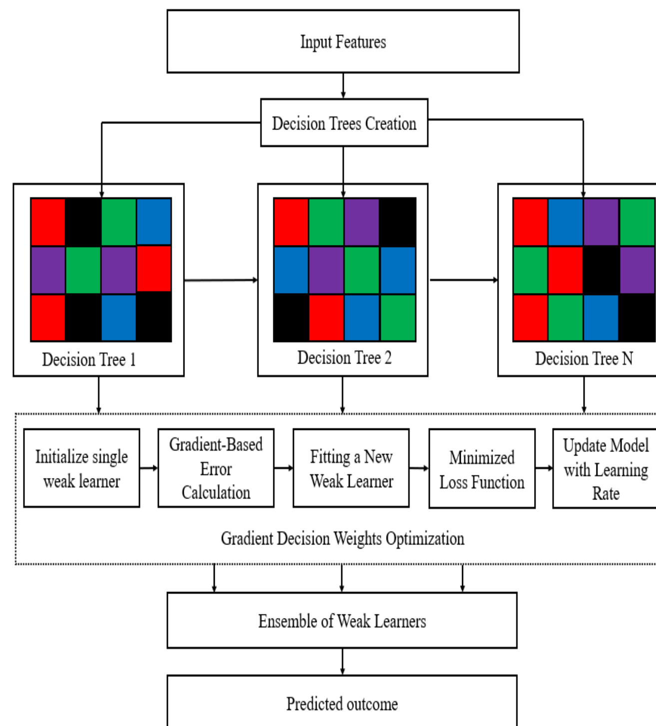

Fig. 3: Proposed GBC Flowchart.

Step 1. Initialization: The process begins with the input features related to software quality. These features could be anything from code complexity metrics (e.g., cyclomatic complexity, lines of code), defect density, developer experience, testing coverage, to project management aspects like team size and communication frequency. The target variable is the software quality category, which falls into one of the six predefined levels. The data is then split into training and testing sets. The training set is used to build the model, and the testing set is used to evaluate its performance.

Step 2. Building the First Weak Learner (Decision Tree 1): Gradient Boosting starts by initializing a simple, often weak, model. In the context of software quality prediction, this is typically a shallow decision tree (a tree with few levels). This initial tree makes a rough prediction for each software project in the training set. The goal is not to get a perfect prediction but to establish a baseline.

Step 3. Computing the Pseudo-Residuals (Gradient-Based Error Calculation): The next crucial step is calculating the pseudo-residuals. These residuals represent the difference between the actual software quality category and the prediction made by the first tree. Since we're dealing with a multi-class classification problem (six categories), the residuals are calculated based on the gradient of a loss function. A common choice for multi-class classification is the categorical cross-entropy loss.

The gradient calculation essentially tells us how much the prediction needs to be adjusted for each project to move closer to the actual category. Projects where the initial tree made a poor prediction will have larger residuals, indicating a greater need for adjustment.

Step 4. Fitting a New Weak Learner (Decision Tree 2): A new decision tree is then built, but this time, it's trained to predict the pseudo-residuals calculated in the previous step, rather than the original software quality categories. This tree learns to correct the errors made by the first tree.

Step 5. Minimizing the Loss Function: The new tree's predictions are combined with the predictions of the first tree. The goal is to minimize the overall loss function, which measures the discrepancy between the combined predictions and the actual software quality categories.

Step 6. Updating the Model with Learning Rate: A learning rate (a small value between 0 and 1) is introduced to control the contribution of the new tree. This learning rate scales down the impact of the new tree's predictions, preventing the model from overfitting to the training data. The model is updated by adding the scaled predictions of the new tree to the predictions of the previous trees.

Step 7. Iterative Process (Decision Trees 3 to N): Steps 3 through 6 are repeated iteratively. In each iteration, a new decision tree is built to predict the residuals of the combined predictions from the previous trees. The model is updated with the scaled predictions of this new tree. This iterative process continues for a predefined number of iterations or until a stopping criterion is met (e.g., when the performance on a validation set stops improving).

Step 8. Ensemble of Weak Learners: The final model is an ensemble of all the decision trees built during the iterations. Each tree contributes to the final prediction, with the later trees focusing on correcting the errors made by the earlier trees.

Step 9. Prediction of Software Quality Category: To predict the software quality category for a new software project, the input features are passed through all the trees in the ensemble. Each tree makes a prediction, and the final prediction is obtained by combining the predictions of all the trees. For multi-class classification, the predictions are typically combined using a softmax function, which outputs the probability of each category. The category with the highest probability is chosen as the final prediction.

5. RESULTS AND DISCUSSION

5.1 Dataset Description

The Software Quality Prediction dataset is a collection of data aimed at predicting the quality of software components based on various metrics and characteristics. This dataset is crucial for researchers and practitioners in the field of software engineering who are focused on improving software reliability, maintainability, and overall quality. The dataset typically includes various attributes related to software code, such as complexity measures, code churn, fault density, and other relevant metrics.

5.2 Results and Description

Figure 3 presents a graphical representation of dataset features, possibly in the form of histograms, bar charts, or scatter plots. This visualization aids users in gaining insights into the distribution and characteristics of the dataset.

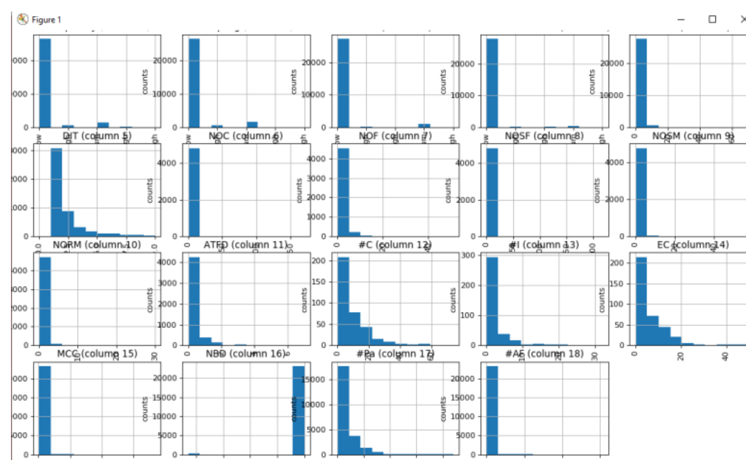

Fig. 3: Graphical representation of Dataset Features.

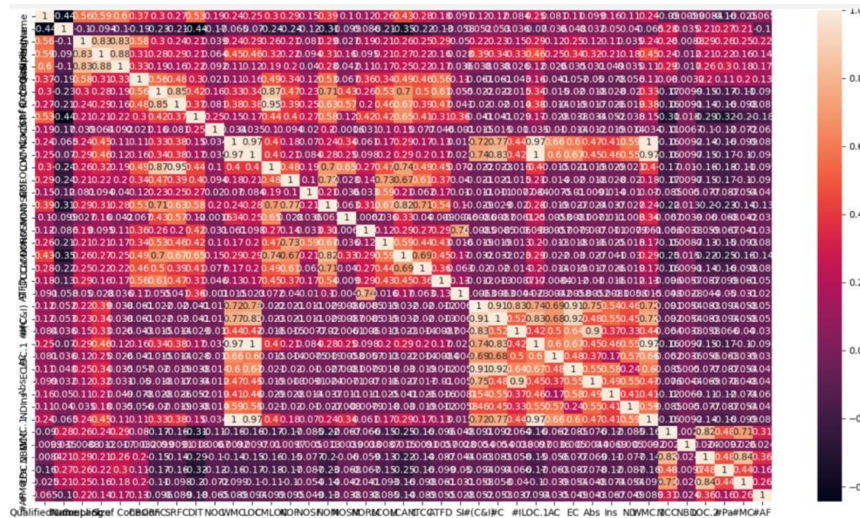

Fig. 4: Data Visualization using Heatmap

Figure 4 showcases data visualization using a heatmap. Heatmaps are effective for displaying correlations between different features in the dataset, providing insights into potential relationships and dependencies.

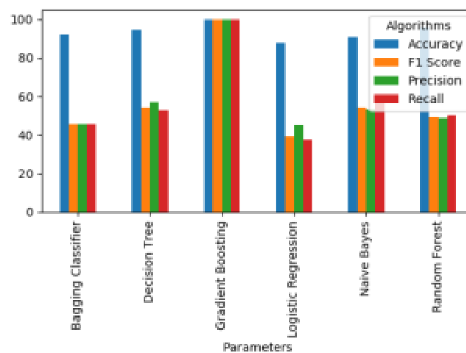

Fig. 5: Graphical representation of Performance matrices

Fig 5 offers a graphical representation of the performance metrics, possibly in the form of bar charts or line graphs. Visualizing the metrics aids users in comparing the performance of different algorithms and making informed decisions in that Xgboost algorithm is best algorithm.

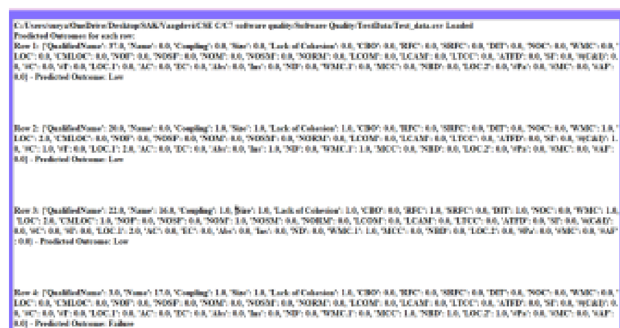

Fig. 6: Model Predication on test data.

5. CONCLUSION

In conclusion, integrating machine learning (ML) techniques into software quality prediction represents a substantial advancement in software engineering. Traditional quality assurance methods,

such as manual code reviews and standard testing procedures, have shown limitations in preemptively identifying and mitigating defects, particularly as software systems grow more complex and development cycles become shorter. ML adoption offers a transformative approach to these challenges by leveraging data-driven insights to forecast potential quality issues before they materialize.

REFERENCES

[1] Chowdhury, Rajarshi Roy, Azam Che Idris, and Pg Emeroylariffion Abas. "A Deep Learning Approach for Classifying Network Connected IoT Devices Using Communication Traffic Characteristics." Journal of Network and Systems Management 31, no. 1 (2023): 26.

[2] Kotak, Jaidip, and Yuval Elovici. "IoT device identification based on network communication analysis using deep learning." Journal of Ambient Intelligence and Humanized Computing 14, no. 7 (2023): 9113-9129.

[3] Koball, Carson, Bhaskar P. Rimal, Yong Wang, Tyler Salmen, and Connor Ford. "IoT Device Identification Using Unsupervised Machine Learning." Information 14, no. 6 (2023): 320.

[4] Elngar, Ahmed, and Adriana Burlea-Schiopoiu. "Feature selection and dynamic network traffic congestion classification based on machine learning for Internet of Things." Wasit Journal of Computer and Mathematics Science 2, no. 2 (2023): 72-86.

[5] Zarzoor, Ahmed R., Nadia Adnan Shiltagh Al-Jamali, and Ibtesam RK Al-Saedi. "Traffic Classification of IoT Devices by Utilizing Spike Neural Network Learning Approach." Mathematical Modelling of Engineering Problems 10, no. 2 (2023).

[6] Elhaloui, Loubna, Sanaa El Filali, Mohamed Tabaa El Habib Benlahmer, Youness Tace, and Nouha Rida. "Machine learning for internet of things classification using network traffic parameters." International Journal of Electrical and Computer Engineering (IJECE) 13, no. 3 (2023): 3449-3463.

[7] Malathi, C., and I. Naga Padmaja. "Identification of cyber attacks using machine learning in smart IoT networks." Materials Today: Proceedings 80 (2023): 25182523.

[8] Senthil Kumaran, S., and S. P. Balakannan. "IoT Capabilities Analysis by Using Optimized Machine Learning with Uncertain Traffic Modeling." Journal of Uncertain Systems 16, no. 01 (2023): 2242005.

[9] Belkadi, Omayma, Alexandru Vulpe, Yassin Laaziz, and Simona Halunga. "ML-Based Traffic Classification in an SDN-Enabled Cloud Environment." Electronics 12, no. 2 (2023): 269.