Predicting Software Quality Using Machine Learning: A Comparative Study

Imtiaz Hossain, Mohammed Asifur Rahman
Department of Computer Science and Engineering
BRAC University, Dhaka, Bangladesh
Email: imtiaz.hossain@g.bracu.ac.bd, mohammed.asifur.rahman@g.bracu.ac.bd

***Abstract*—Software quality prediction is crucial for efficient resource allocation and timely delivery of reliable software products. This paper presents a comprehensive comparative analysis of machine learning approaches for predicting software quality based on various code metrics. We evaluate neural networks, k-nearest neighbors, decision trees, and random forest classifiers on a dataset containing 1600 samples with 9 features. Our experimental results demonstrate that the decision tree classifier achieves the best performance with a weighted F1-score of 0.4418. We also explore k-means clustering as an unsupervised approach to identify patterns in code quality. The findings provide insights into the effectiveness of different machine learning techniques for software quality prediction and highlight the challenges in this domain.**

***Index Terms*—Software quality prediction, machine learning, neural networks, decision trees, random forest, k-means clustering, code metrics**

I. Introduction

Software quality assurance is a critical aspect of the software development lifecycle, directly impacting maintenance costs, user satisfaction, and system reliability. Traditional manual code review processes are time-consuming and often inconsistent. Automated approaches for predicting software quality can significantly reduce effort while improving consistency in quality assessment.

Machine learning techniques have shown promising results in various software engineering tasks, including defect prediction, code smell detection, and quality assessment. However, selecting the appropriate algorithm and evaluating its effectiveness remains challenging due to the complex nature of software metrics and their relationships with quality attributes.

This paper presents a comparative study of supervised and unsupervised machine learning approaches for software quality prediction. We evaluate neural networks, k-nearest neighbors (KNN), decision trees, and random forests on a dataset of code metrics, comparing their performance using various evaluation metrics. Additionally, we explore k-means clustering to identify inherent patterns in the data without prior labeling.

The main contributions of this work are:

- A comprehensive analysis of code metrics and their relationships with software quality
- Comparative evaluation of multiple machine learning models for quality prediction
- Application of unsupervised learning for pattern discovery in code quality
- Identification of the most effective model for this prediction task

II. Dataset Description

The dataset used in this study consists of 1600 samples with 9 features representing various code metrics. The features include:

- **Lines of Code**: Number of lines in the code module
- **Cyclomatic Complexity**: Measure of code complexity
- **Number of Functions**: Count of functions in the module
- **Code Churn**: Measure of recent changes to the code
- **Comment Density**: Ratio of comments to code
- **Number of Bugs**: Count of identified defects
- **Has Unit Tests**: Binary indicator of test coverage
- **Code Owner Experience**: Measure of developer experience
- **Quality Label**: Target variable with three classes (High, Medium, Low)

The dataset exhibits some missing values in the Lines of Code, Code Churn, and Comment Density features (80 missing values each). The target variable distribution is relatively balanced with 566 High-quality, 533 Low-quality, and 501 Medium-quality samples.

TABLE I: Descriptive Statistics of Numerical Features

Feature	Count	Mean	Std	Min	Max
Lines_of_Code	1520	4939.27	2867.25	106.00	9998.00
Cyclomatic_Complexity	1600	25.08	13.88	1.00	49.00
Num_Functions	1600	103.18	55.50	5.00	199.00
Code_Churn	1520	102.57	50.55	-64.28	295.14
Comment_Density	1520	0.55	0.26	0.10	1.00
Num_Bugs	1600	2.93	1.72	0.00	10.00
Code_Owner_Experience	1600	5.05	2.56	1.00	9.00

III. Data Analysis

We performed exploratory data analysis to understand the distribution of features and their relationships with the target variable. Figure 1 shows the distribution of quality labels, which is relatively balanced across the three categories.

The histogram analysis of numerical features (Figure 2) revealed diverse distributions across different metrics. Some features like Lines of Code and Number of Functions showed approximately normal distributions, while others like Comment Density exhibited right-skewed distributions.

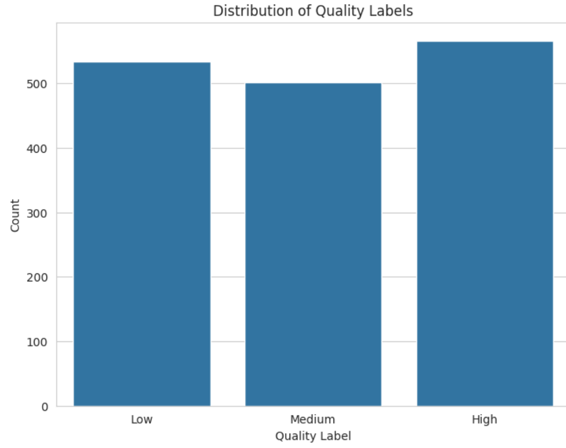


Fig. 1: Distribution of Quality Labels

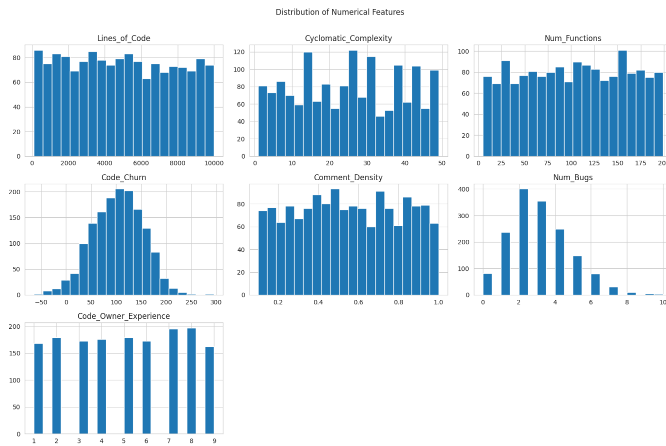


Fig. 2: Distribution of Numerical Features

The correlation analysis (Figure 3) revealed several interesting relationships between features. Notably, Lines of Code showed strong positive correlation with Cyclomatic Complexity and Number of Functions, indicating that larger code modules tend to be more complex and contain more functions. Number of Bugs showed moderate positive correlation with Cyclomatic Complexity and Lines of Code, suggesting that more complex and larger code modules tend to have more defects.

IV. Dataset Pre-processing

Several pre-processing steps were applied to prepare the data for modeling:

A. Handling Missing Values

The dataset contained missing values in three features: Lines of Code, Code Churn, and Comment Density (80 missing values each). To address this, we employed imputation using the most frequent value (mode) for each respective column. This method was implemented using Scikit-learn's `SimpleImputer` and was chosen to preserve the original distribution of the data without introducing artificial values that might result from other techniques like mean or median imputation.

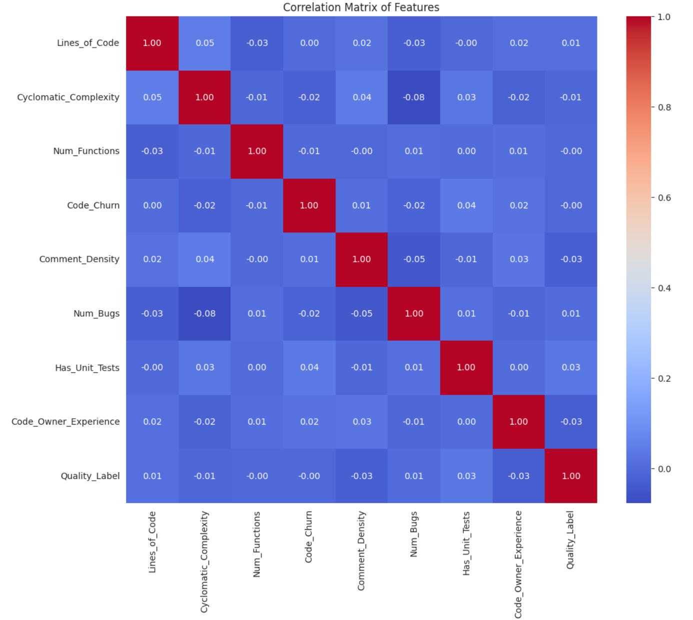


Fig. 3: Correlation Matrix of Features

B. Handling Outliers and Data Transformation

Analysis of the feature distributions revealed the presence of outliers, particularly in the `Code_Churn` and `Num_Functions` columns. To mitigate the influence of these extreme values on the models, we applied Winsorizing. This technique capped the values at the 1st and 99th percentiles, effectively replacing the most extreme high and low values with less extreme ones. Additionally, the `Code_Churn` feature contained negative values, which represent lines of code being removed. To ensure the feature consistently represents the magnitude of change, we transformed it by taking its absolute value.

C. Encoding Categorical Variables

The binary feature 'Has Unit Tests' was encoded with 1 representing 'Yes' and 0 representing 'No'. The target variable 'Quality Label' was encoded with numerical values: 0 for High, 1 for Low, and 2 for Medium quality.

D. Feature Scaling

All features were standardized using StandardScaler to have zero mean and unit variance. This preprocessing step is particularly important for distance-based algorithms like KNN and gradient-based optimization in neural networks.

E. Data Splitting

The dataset was split into training and testing sets with an 90:10 ratio, maintaining the class distribution through stratified sampling.

V. Model Training

We implemented and compared four supervised learning models and one unsupervised approach:

A. Neural Network

A fully connected neural network was implemented with the following architecture:

- Input layer: 8 neurons (matching the number of features)
- Hidden layer 1: 64 neurons with ReLU activation and dropout (0.3)
- Hidden layer 2: 32 neurons with ReLU activation and dropout (0.3)
- Output layer: 3 neurons with softmax activation (for multi-class classification)

The model was trained for 50 epochs using the Adam optimizer with categorical cross-entropy loss.

B. K-Nearest Neighbors (KNN)

A KNN classifier was implemented with k=5 neighbors. This instance-based learning algorithm classifies samples based on the majority class among their k nearest neighbors in feature space.

C. Decision Tree

A decision tree classifier was implemented with default parameters. Decision trees learn hierarchical decision rules based on feature values to classify instances.

D. Random Forest

A random forest classifier, which is an ensemble learning method, was also implemented. This model constructs a multitude of decision trees at training time and outputs the class that is the mode of the classes of the individual trees.

E. Determining Optimal Clusters with Elbow Method

To determine the optimal number of clusters for k-means clustering, we employed the elbow method. This method involves running k-means clustering on the dataset for a range of values of k (number of clusters) and calculating the sum of squared distances (inertia) from each point to its assigned cluster center. As shown in Figure 4, we plotted the inertia values for k ranging from 1 to 9. The ”elbow” of the plot—the point where the rate of decrease sharply shifts—suggests the optimal number of clusters. In our analysis, the elbow appears to be at k=3, which aligns with the three quality categories in our dataset.

F. K-Means Clustering (Unsupervised)

We applied k-means clustering with k=3 (as determined by the elbow method) to explore natural groupings in the data without using quality labels. The results showed some alignment with the actual quality labels, but with significant overlap between clusters.

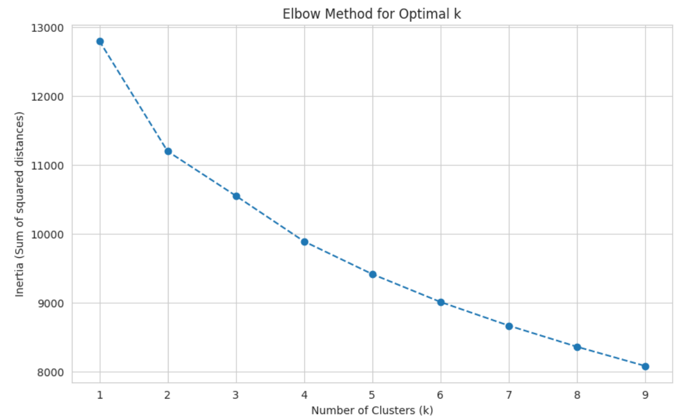


Fig. 4: Elbow Method for Determining Optimal k

TABLE II: Performance Comparison of Models (Weighted Averages)

Model	Accuracy	Precision	Recall	F1-Score
Neural Network	0.35	0.35	0.35	0.35
KNN	0.31	0.30	0.31	0.30
Decision Tree	0.44	0.44	0.44	0.44
Random Forest	0.36	0.36	0.36	0.35

VI. Model Evaluation

We evaluated model performance using multiple metrics including accuracy, precision, recall, and F1-score. The models were compared on a held-out test set.

The decision tree classifier achieved the best performance with an accuracy and weighted F1-score of approximately 0.44. The other models performed less effectively, with the KNN model showing the lowest scores.

Figure 6 shows the confusion matrices for all models. The decision tree showed more balanced performance across classes compared to the other models, which exhibited stronger biases toward certain classes.

The ROC curve analysis (Figure 7) showed that all models achieved similar performance in terms of micro-average AUC, with values around 0.60. This indicates moderate discrimination power for all classifiers.

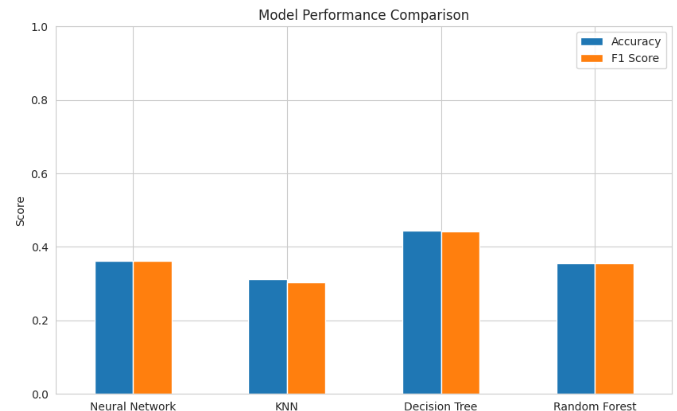
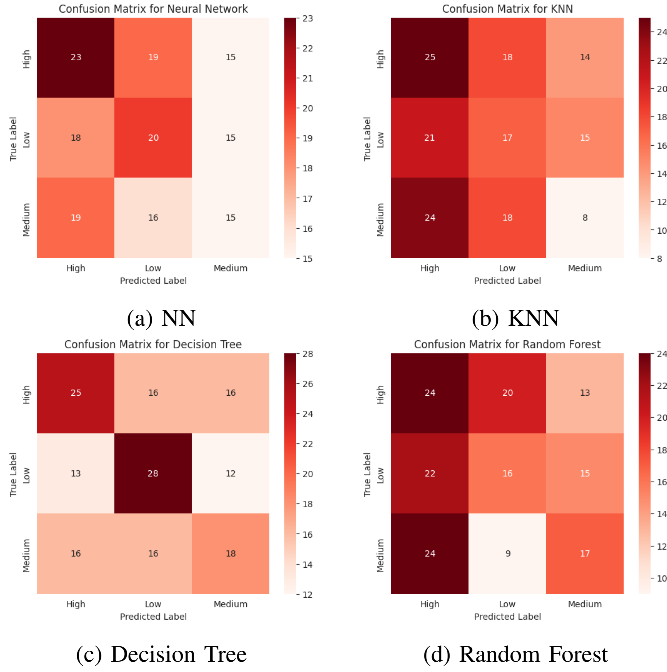


Fig. 5: Model Performance Comparison



(a) NN (b) KNN

(c) Decision Tree (d) Random Forest

Fig. 6: Confusion Matrices for All Models

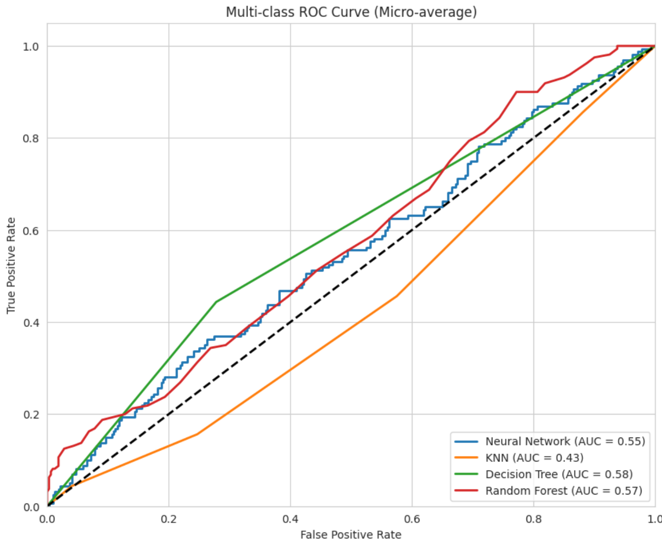


Fig. 7: ROC Curves for All Models (Micro-average)

A. K-Means Clustering Visualization

To visualize the results of our k-means clustering analysis, we applied Principal Component Analysis (PCA) to reduce the dimensionality of our feature space to two dimensions while preserving as much variance as possible. Figure 8 shows the data points colored by their cluster assignments, with cluster centers marked. The visualization reveals that while some separation exists between clusters, there is significant overlap, suggesting that the code metrics alone may not be sufficient to clearly distinguish between different quality categories without supervision.

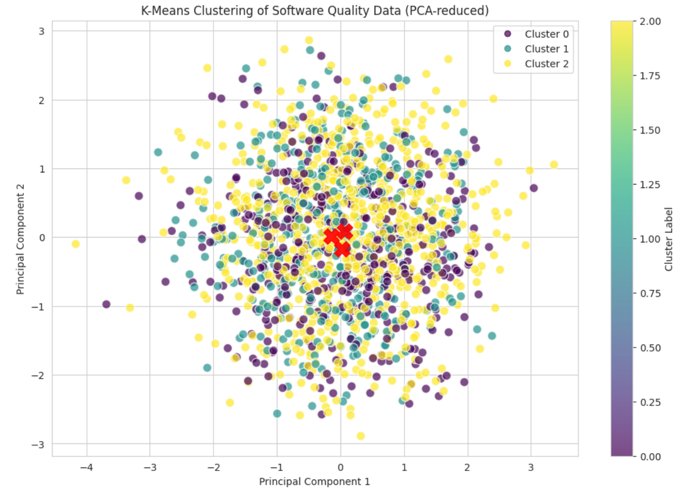


Fig. 8: K-Means Clustering Results (PCA-reduced)

VII. Conclusion

This study compared the performance of neural networks, k-nearest neighbors, decision trees, and random forests for software quality prediction. Among these models, the decision tree classifier achieved the best performance with a weighted F1-score of 0.4418.

The relatively modest performance of all models suggests that predicting software quality from code metrics alone is challenging. The complex, non-linear relationships between code metrics and quality attributes may require more sophisticated feature engineering or alternative modeling approaches.

Our k-means clustering analysis with the elbow method confirmed that k=3 is an appropriate choice for the number of clusters, corresponding to the three quality categories. However, the PCA visualization of clustering results showed significant overlap between clusters, further supporting the challenge of distinguishing software quality based solely on code metrics.

Several factors may contribute to the performance limitations:

- The inherent complexity and subjectivity in defining software quality
- Potential missing features that better capture quality aspects
- Noisy or inconsistent labeling of quality categories
- Non-linear relationships that are difficult to capture with the implemented models

Future work could explore more advanced techniques such as other ensemble methods (e.g., Gradient Boosting), different neural network architectures, or incorporating additional data sources such as version history or developer activity. Feature engineering approaches that create more informative representations of code characteristics might also improve prediction performance.

Despite the challenges, machine learning approaches show promise for automated quality assessment and could be valuable tools for supporting human experts in quality evaluation tasks.

REFERENCES

[1] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in Proceedings of the 27th International Conference on Software Engineering, 2005, pp. 284–292.