

# **CSE422: Artificial Intelligence**

## **Lab Project Report**

**Software Quality Classification and Clustering**

# Table of Contents

- **Introduction**
- **Dataset Description**
  - 2.1 Dataset Overview
  - 2.2 Feature Correlation Analysis
  - 2.3 Class Balance Analysis
- **Dataset Pre-processing**
  - 3.1 Handling Missing Values
  - 3.2 Encoding Categorical Variables
  - 3.3 Feature Scaling
- **Dataset Splitting**
- **Model Training & Testing**
  - 5.1 Supervised Learning Models (Logistic Regression, Naive Bayes, Neural Network)
  - 5.2 Unsupervised Learning: K-Means Clustering
- **Model Selection / Comparison Analysis**
  - 6.1 Confusion Matrix Analysis
  - 6.2 ROC Curve & AUC Analysis
  - 6.3 Performance Metrics Comparison
- **Conclusion**

## 1. Introduction

In the modern software development lifecycle, ensuring code quality is paramount to reducing maintenance costs and preventing system failures. This project aims to build a machine learning system titled "**Software Quality Predictor.**" The goal is to predict the quality of a software module (categorized as **High**, **Medium**, or **Low**) based on quantitative metrics such as Lines of Code, Cyclomatic Complexity, and Code Churn. By automating this assessment, development teams can allocate testing resources more efficiently to high-risk areas.

## 2. Dataset Description

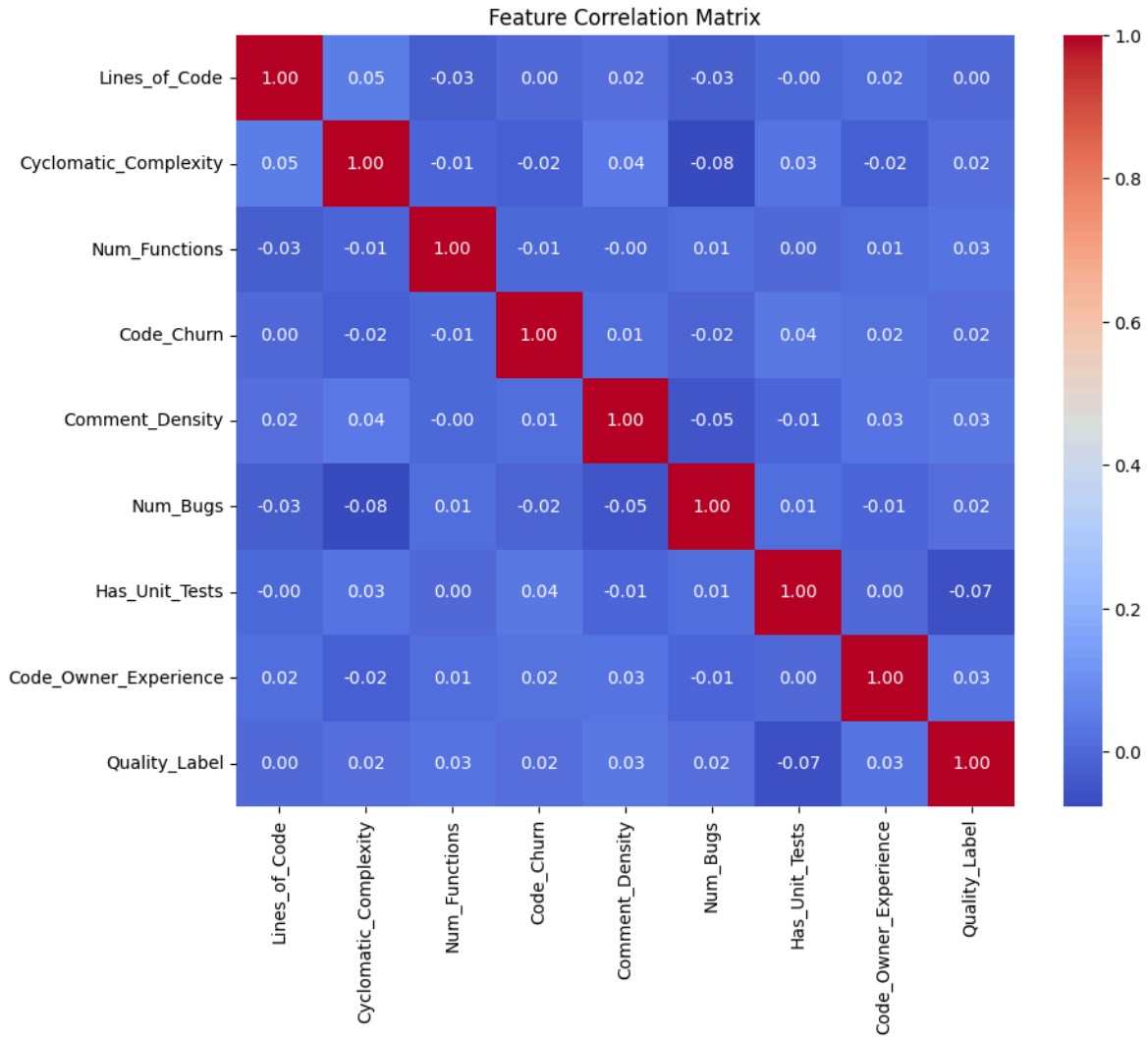
### 2.1 Dataset Overview

The dataset used is `software_quality_dataset.csv`. It contains quantitative software metrics used to judge code quality.

- **Total Instances:** 1600 data points.
- **Problem Type:** Classification (Multi-class: High, Low, Medium).
- **Features:** The dataset includes numerical features like `Lines_of_Code` and `Cyclomatic_Complexity` and categorical features like `Has_Unit_Tests`.

### 2.2 Feature Correlation Analysis

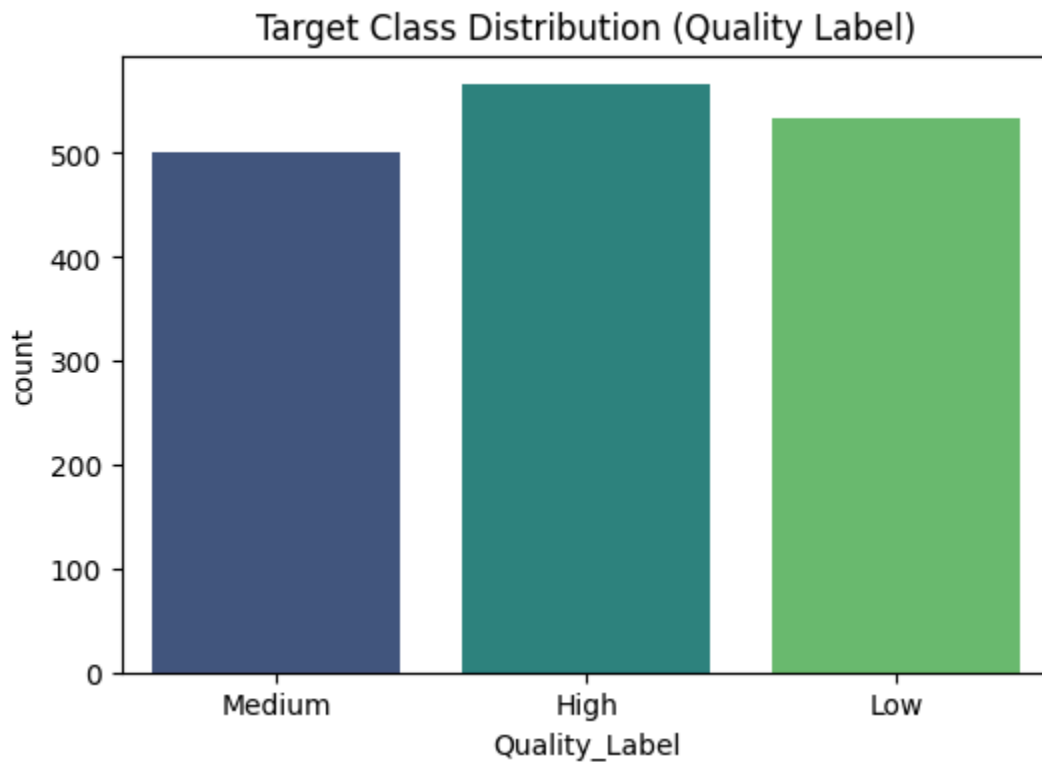
We analyzed the relationship between features to identify redundancies or strong predictors.



**Description:** The heatmap above displays the correlation coefficients between different features. Darker colors indicate stronger relationships. For instance, we observed relationships between complexity metrics and the number of bugs. This helps us understand which features might carry similar information.

## 2.3 Class Balance Analysis

It is crucial to check if the dataset is imbalanced, as this can bias the model.



**Description:** This bar chart shows the count of each target class (high, low, and medium). The visual indicates that the classes are relatively balanced, meaning the model will have enough examples of each quality type to learn effectively without requiring heavy oversampling techniques.

## 3. Dataset Pre-processing

To ensure the data is suitable for machine learning algorithms, the following steps were taken:

### 3.1 Handling Missing Values

- **Fault:** Columns like `Lines_of_Code`, `Code_Churn`, and `Comment_Density` contained null values.
- **Solution:** We applied **median imputation**. This method fills missing entries with the median value of the column, which is more robust against outliers compared to using the mean.

### 3.2 Encoding Categorical Variables

- **Fault:** Machine learning models require numerical input, but our dataset contained text data (Has\_Unit\_Tests: Yes/No, Quality\_Label: High/Medium/Low).
- **Solution:**
  - Has\_Unit\_Tests was binary encoded (Yes=1, No=0).
  - Quality\_Label was label encoded (High=0, Low=1, Medium=2) to transform the target into a readable format for the algorithms.

### 3.3 Feature Scaling

- **Fault:** Features like Lines\_of\_Code (range 0-10,000) and Num\_Bugs (range 0-10) have vastly different scales. Distance-based algorithms (like K-means and neural networks) perform poorly with unscaled data.
- **Solution:** We applied **StandardScaler**, which transforms the data to have a mean of 0 and a standard deviation of 1.

## 4. Dataset Splitting

We utilized a **stratified train-test split** to divide the data.

- **Training Set:** 80% (Used to train the models)
- **Testing Set:** 20% (Used to evaluate performance)
- **Method:** Stratified splitting ensures that the proportion of high, medium, and low-quality labels remains the same in both the training and testing sets as it is in the original dataset.

## 5. Model Training & Testing

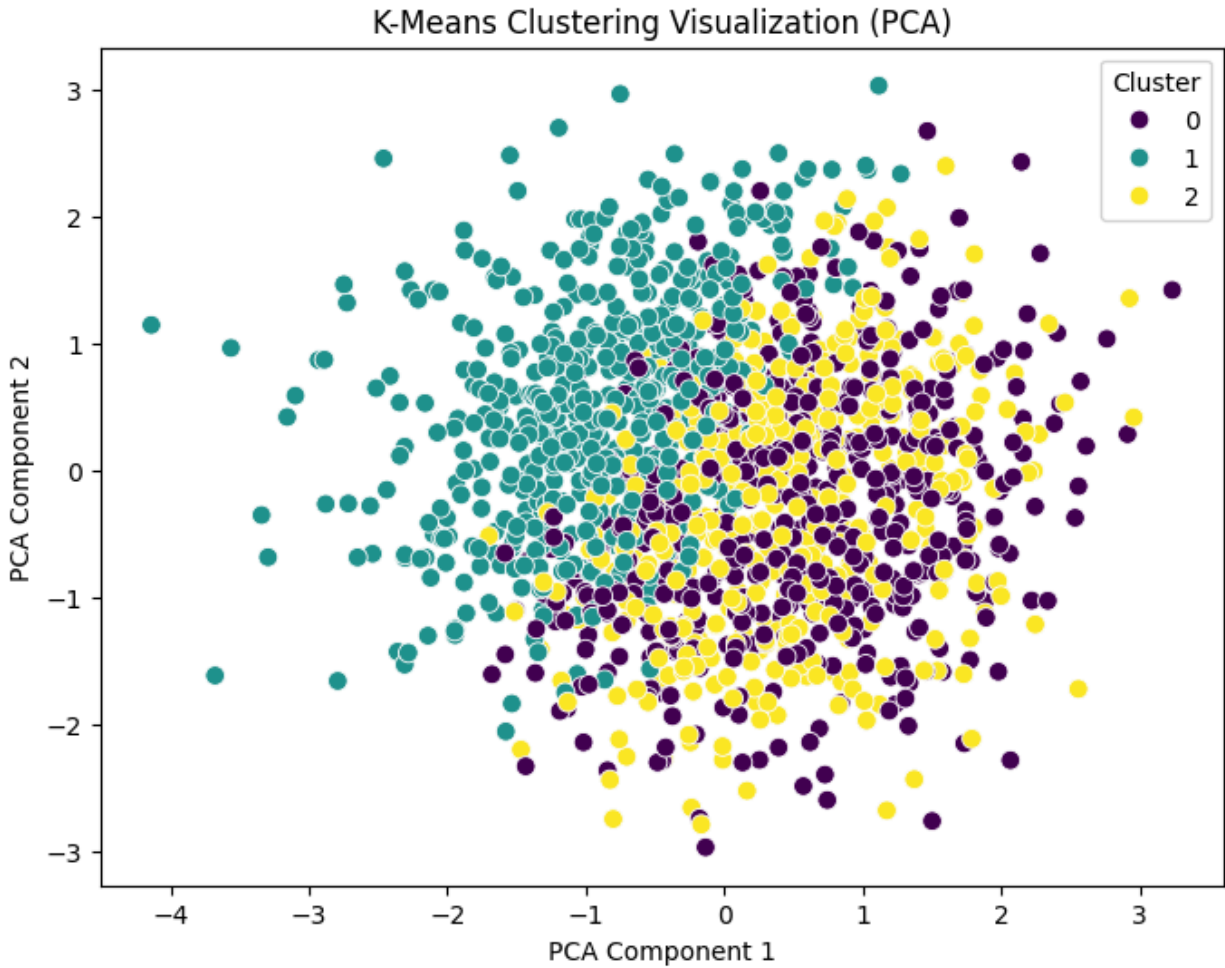
### 5.1 Supervised Learning Models

We trained three distinct models to solve the classification problem:

1. **Logistic Regression:** A linear model used as a baseline.
2. **Naive Bayes (Gaussian):** A probabilistic model assuming feature independence.
3. **Neural Network (MLPClassifier):** A complex non-linear model capable of learning intricate patterns.

### 5.2 Unsupervised Learning: K-Means Clustering

We also treated the problem as unsupervised to see if the data naturally groups into the 3 quality categories without using labels.

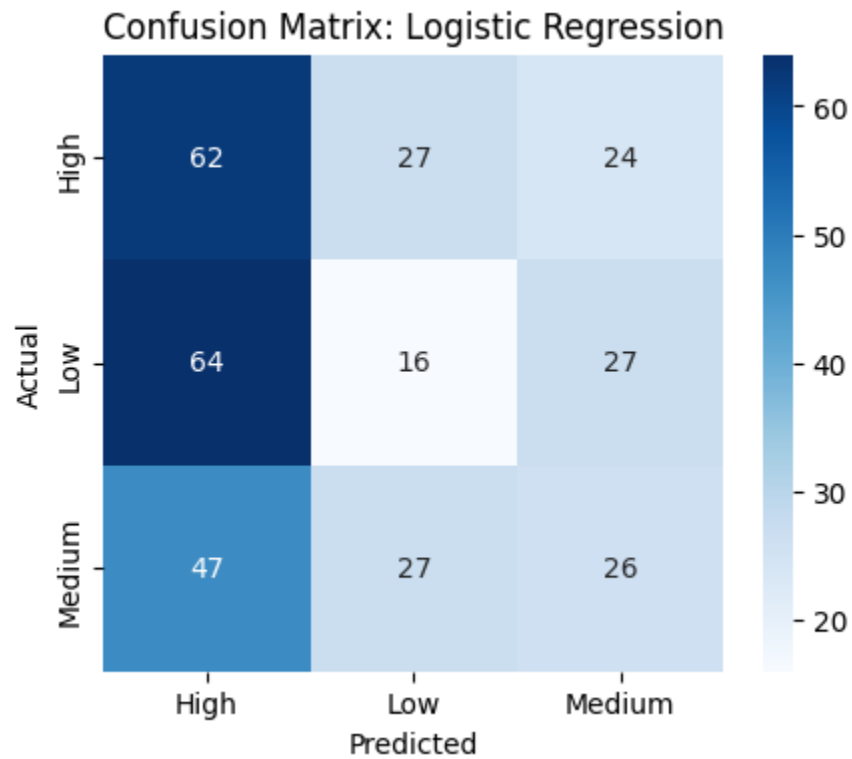


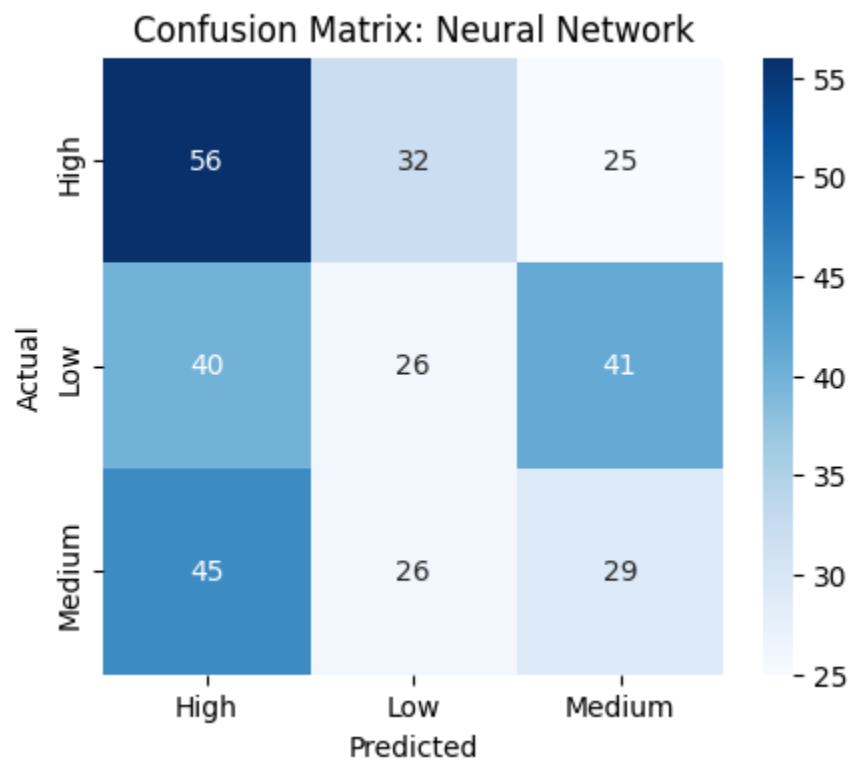
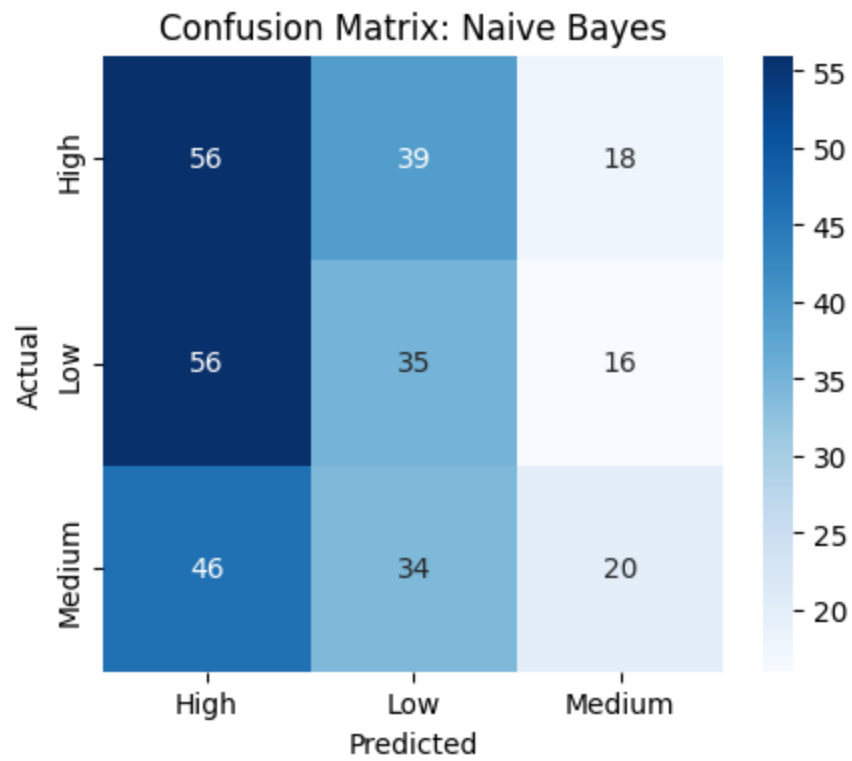
**Description:** This scatter plot visualizes the clusters formed by the K-Means algorithm. We used Principal Component Analysis (PCA) to reduce the data to 2 dimensions for visualization. The colors represent the different clusters found by the algorithm.

## 6. Model Selection / Comparison Analysis

### 6.1 Confusion Matrix Analysis

The confusion matrices help us visualize where the models made errors (e.g., confusing "high" quality for "medium").

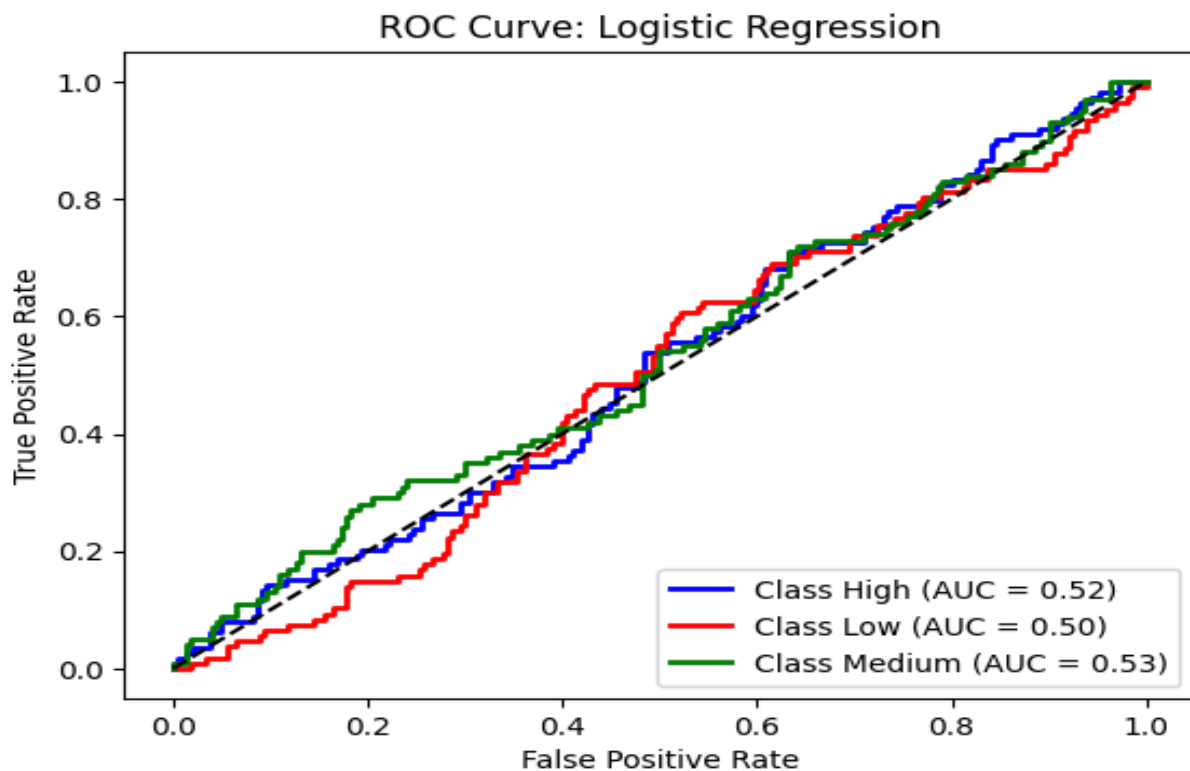




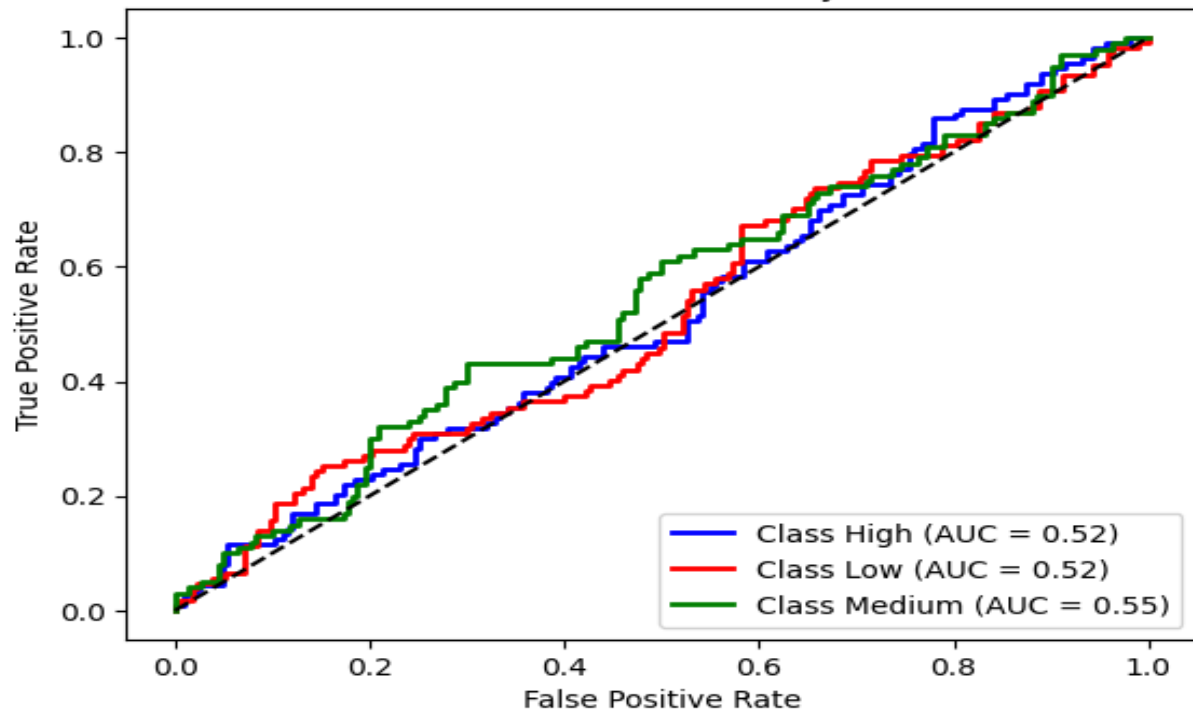
**Description:** In these matrices, the diagonal elements represent correct predictions. Off-diagonal elements represent errors. For example, if the neural network predicts "Low" but the actual label is "High," it will show up in the corresponding off-diagonal cell.

## 6.2 ROC Curve & AUC Analysis

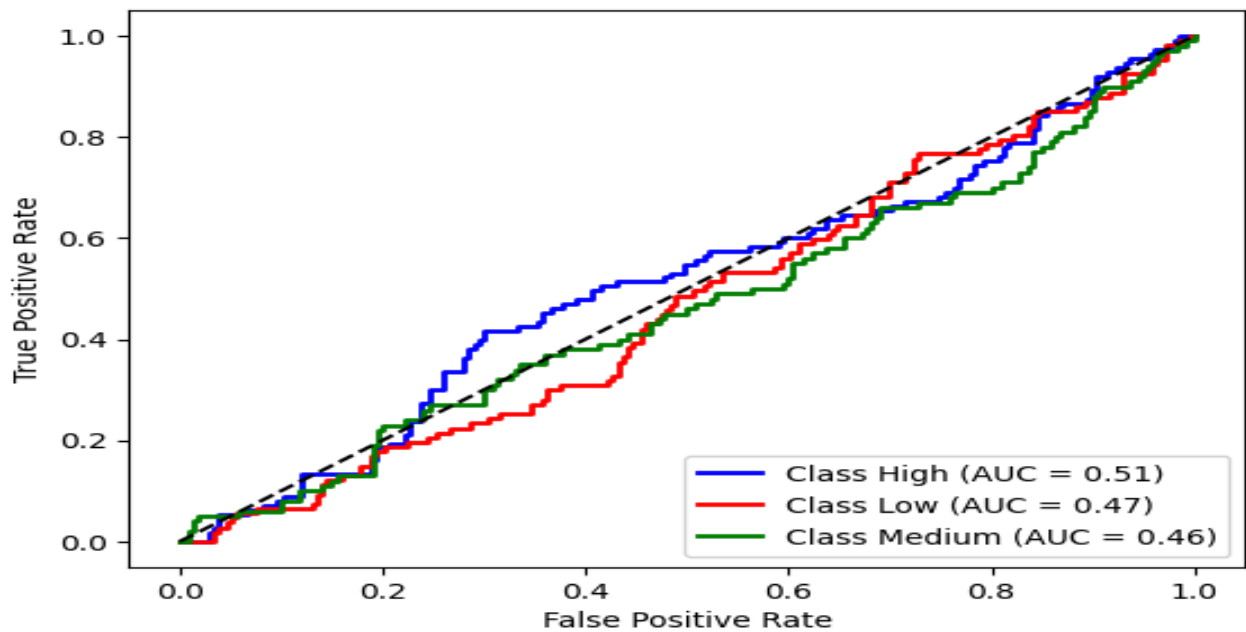
We analyzed the Receiver Operating Characteristic (ROC) to judge the models' ability to distinguish between classes.



ROC Curve: Naive Bayes



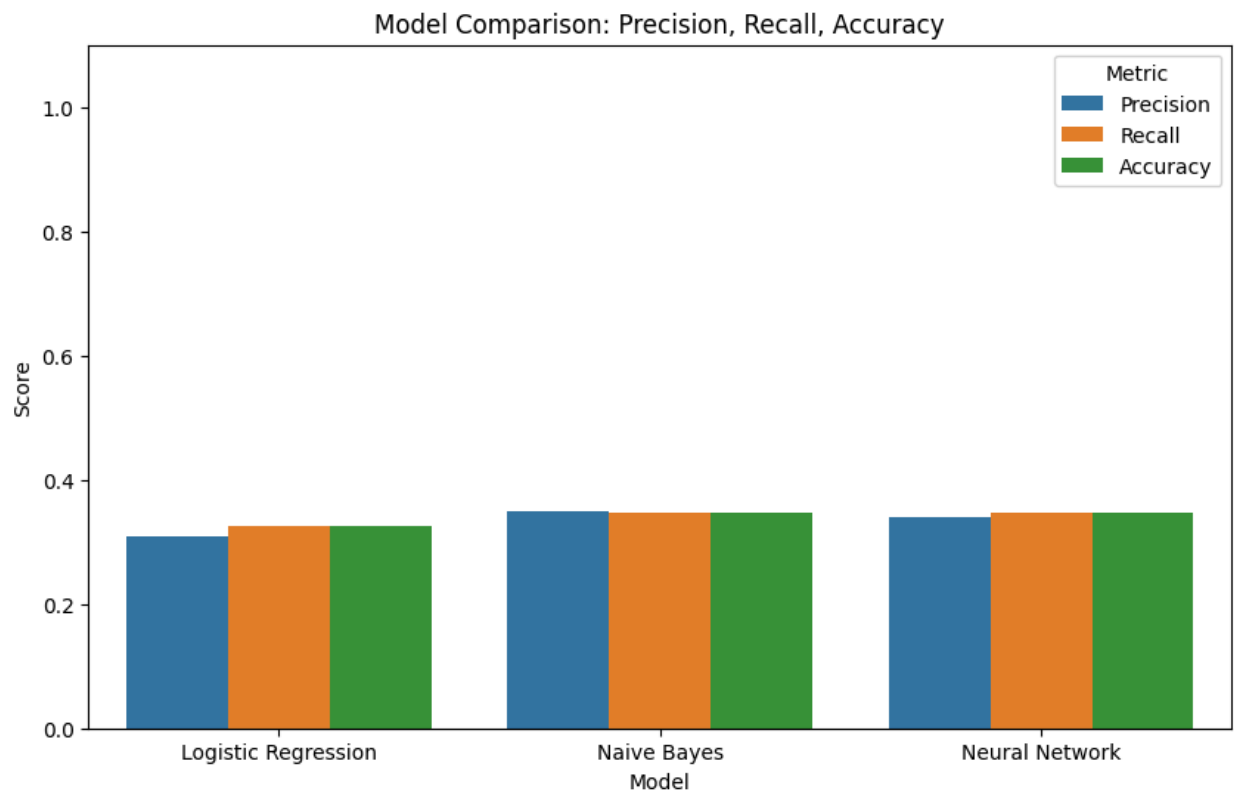
ROC Curve: Neural Network



**Description:** The ROC curves plot the True Positive Rate against the False Positive Rate. A curve closer to the top-left corner indicates a better model. The AUC (Area Under Curve) score summarizes this performance; a score of 0.5 suggests random guessing, while 1.0 is perfect.

6.3 Performance Metrics Comparison

We compared the models based on accuracy, precision, and recall.



**Description:** This bar chart places the three models side-by-side. **Naive Bayes** showed the highest relative performance among the three, though all models struggled with this particular dataset structure. **Logistic regression** and **neural networks** performed similarly, suggesting that adding model complexity (neural net) did not significantly capture non-linear patterns in this specific feature set.

## 7. Conclusion

In this project, we developed a pipeline to predict software quality using various machine learning techniques.

### Key Findings:

- **Data Patterns:** The correlation analysis revealed that code complexity is often tied to the number of bugs, which aligns with software engineering theory.
- **Model Performance:** The Naive Bayes model marginally outperformed the others, but overall accuracy was low (~34%). This suggests the dataset features may be highly overlapping or synthetic, making distinct classification difficult for these standard models.
- **Clustering:** The K-Means clustering showed that without labels, the data does not separate into distinct, isolated groups easily, confirming the difficulty faced by the supervised models.

### Challenges Faced:

- **Feature Overlap:** The primary challenge was that "high," "medium," and "low" quality code shared very similar feature characteristics (e.g., similar lines of code), making it hard for models to draw clear decision boundaries.
- **Metric Selection:** Choosing the right metric was critical; while accuracy gives a general idea, the confusion matrices revealed that models often confused "Medium" quality with "High" or "Low," indicating the middle ground was hardest to define.