Predicting Code Quality: Comparative Analysis of ML models and Multi-Task GNNs

A Project Report Submitted

in Partial Fulfilment of the Requirements

for the Degree of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

by

Maripally Amogh - 2022BCS0022

Yenika Surendra Kumar Reddy - 2022BCS0033

Mohammad Haseeb - 2022BCS0107

Abhishek Akash - 2022BCS0002

to

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF INFORMATION TECHNOLOGY

KOTTAYAM-686635, INDIA

November 2025

DECLARATION

We, **Maripally Amogh** (**Roll No: 2022BCS0022**), **Yenika Surendra Kumar Reddy** (**Roll No: 2022BCS0033**), **Mohammad Haseeb** (**Roll No: 2022BCS0107**), **Abhishek Akash** (**Roll No: 2022BCS0002**), hereby declare that this report entitled **"Predicting Code Quality: Comparative Analysis of ML models and Multi-task GNNs"** submitted to Indian Institute of Information Technology Kottayam towards partial requirement of **Bachelor of Technology** in **Computer Science and Engineering** is an original work carried out by us under the supervision of **Dr. Sara Renjit** and has not formed the basis for the award of any degree or diploma elsewhere. We have upheld academic ethics and honesty, and external information has been duly acknowledged.

Kottayam-686635	**Maripally Amogh** (2022BCS0022)
November 2025	**Yenika Surendra Kumar Reddy** (2022BCS0033)
	Mohammad Haseeb (2022BCS0107)
	Abhishek Akash (2022BCS0002)

CERTIFICATE

This is to certify that the work contained in this project report entitled **"Predicting Code Quality: Comparative Analysis of ML models and Multi-Task GNNs"** submitted by **Maripally Amogh** (2022BCS0022), **Yenika Surendra Kumar Reddy** (2022BCS0033), **Mohammad Haseeb** (2022BCS0107), **Abhishek Akash** (2022BCS0002) to the Indian Institute of Information Technology Kottayam towards partial requirement of **Bachelor of Technology** has been carried out by them under my supervision and has not been submitted elsewhere for any degree.

Kottayam-686635
November 2025

Dr. Sara Renjit
Project Supervisor

ABSTRACT

This project presents an automated framework for evaluating software code quality by comparing traditional machine learning methods with modern representation learning techniques. Conventional approaches depend on static metrics or handcrafted textual features, which capture surface-level characteristics but often miss deeper semantic information within the code.

In this phase, two categories of models are explored: classical ML models using TF–IDF features, and transformer-based models using CodeBERT. A clean preprocessing pipeline is implemented to ensure leak-free data splitting and consistent feature preparation. Accuracy is used as the main evaluation measure to compare performance across methods.

The results show that conventional ML models struggle to understand semantic patterns, especially when code involves complex logic or context dependencies. In contrast, CodeBERT provides richer contextual representations and demonstrates noticeably improved performance due to its ability to capture syntax and semantics more effectively.

Future work will extend this study by integrating Graph Neural Networks (GNNs) to represent structural properties of code, such as relationships captured through abstract syntax trees and control-flow graphs. This will help combine semantic and structural learning for a more complete and reliable code quality assessment framework.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Software quality plays a crucial role in ensuring reliability, maintainability, and long-term scalability of modern systems. As software projects grow in size and complexity, evaluating the quality of source code manually becomes impractical. Traditional approaches to code quality assessment rely on predefined static metrics, code smells, or manual reviews — methods that often fail to capture the deeper semantic and contextual characteristics of code.

With the rise of machine learning (ML) and natural language processing (NLP) techniques, new opportunities have emerged for automating the evaluation of code quality. Early research predominantly used text-based representations of code, such as TF-IDF or bag-of-words, coupled with classical ML classifiers like Logistic Regression, Random Forest, and Gradient Boosting. While these models achieved reasonable predictive power, they were limited by their inability to understand the structural and semantic dependencies within source code.

To overcome these limitations, recent studies have introduced representation learning methods that treat code as a structured form of natural language. In this phase, we explore two complementary approaches for assessing code quality: (i) traditional ML models that rely on TF-IDF representations and statistical learning, and (ii) advanced deep learning models such as CodeBERT, which are pre-trained on large corpora of code and natural language to capture semantic and syntactic information more effectively.

The present phase focuses primarily on the machine learning approach, where TF-IDF features are compared with contextual embeddings generated by a fine-tuned CodeBERT model. This comparative study aims to identify how the inclusion of semantic understanding through pre-trained embeddings improves the reliability and generalization of code quality prediction. The groundwork established in this phase will support future exploration into graph neural network (GNN) architectures for structure-aware code representation and analysis.

1.2 Goal

The primary goal of this study is to research code quality using data-driven methods. This phase emphasizes a comparative study between classical ML models using TF-IDF features and a fine-tuned CodeBERT model leveraging contextual embeddings. The objective is to determine how each approach captures code semantics and contributes to accurate quality assessment across different code granularities.

Specifically, this phase focuses on:

1. Building a clean, consistent dataset pipeline for code samples with well-defined quality labels.
2. Implementing and comparing ML models such as Logistic Regression, Random Forest, Gradient Boosting, and Support Vector Classifier trained on TF-IDF features.
3. Fine-tuning CodeBERT for classification and regression tasks to evaluate improvements gained from contextual code representations.
4. Conducting a leak-free experimental setup with stratified data splits and k-fold validation to ensure robust evaluation.
5. Analyzing performance metrics such as accuracy and ROC-AUC to establish baselines and understand model behavior.
6. Using insights from this phase to guide the design of GNN-based methods for future phases, enabling more structural and relational understanding of source code.

Through this analysis, the study aims to establish a foundation for a comprehensive framework that integrates text-based, contextual, and structure-aware representations for fine-grained code quality assessment.

1.3 Motivation

As software development evolves, codebases increasingly span multiple contributors, languages, and frameworks, making manual code reviews insufficient to maintain quality standards. Poorly structured or semantically inconsistent code not only reduces maintainability but also increases the likelihood of bugs, technical debt, and long-term project instability. Traditional static analysis tools and rule-based metrics provide limited insight, as they often fail to capture the contextual relationships and semantics underlying source code constructs.

The motivation behind this project stems from the need for intelligent, automated systems that can evaluate code quality in a more holistic and data-driven manner. By comparing TF-IDF based ML models with deep contextual models like CodeBERT, we aim to identify the trade-offs between interpretability and semantic richness in automated code analysis. This exploration serves as the foundation for developing advanced hybrid approaches that can balance the precision of statistical learning with the contextual depth of representation learning.

Furthermore, as the field progresses toward graph-based representations and neural models capable of understanding syntax trees and control flow graphs, this project's outcomes will provide empirical and conceptual groundwork for integrating such architectures in future phases.

Chapter 2

Literature Review

2.1 Existing Works

Assessing software code quality has long been a topic of research, with early methods relying on static metrics and rule-based analysis. These approaches—such as cyclomatic complexity [7], coupling and cohesion metrics [2], and maintainability indices—provided interpretable indicators but struggled to scale with the increasing size and complexity of modern, multi-module software systems. As codebases continued to expand, researchers shifted toward machine learning (ML) and deep learning methods capable of capturing subtle semantic and structural patterns that traditional metrics could not identify.

Alon et al. introduced *code2vec* [1], one of the earliest neural models to represent source code through path-based embeddings extracted from Abstract Syntax Trees (ASTs). Their method demonstrated that AST paths effectively encode syntactic and semantic relationships in source code, outperforming traditional handcrafted features in tasks such as method name prediction. Building on this direction, Feng et al. proposed *CodeBERT* [5], a bimodal pre-trained transformer that jointly learns from natural language and programming language tokens, significantly improving performance in semantic code search, clone detection, and defect prediction.

Several works have focused specifically on defectiveness and maintainability prediction. Buse and Weimer [3] developed a readability metric derived from human-annotated code samples, demonstrating strong correlations with maintainability and developer comprehension. Kumar and Sureka [6] expanded this line of work by applying deep learning models directly on code metrics and textual content to estimate maintainability, showing that neural representations capture high-level design complexity. Likewise, White et al. [10] established the value of deep learning for code clone detection, emphasizing that learned representations outperform manually engineered metrics, marking a broader shift in software engineering research toward representation learning.

More recently, graph-based techniques have gained traction due to their ability to capture program structure. Zeng et al. introduced *Tailor* [11], a framework that utilizes multi-relational Code Property Graphs (CPGs)—combining ASTs, Control Flow Graphs (CFGs), and Data Flow Graphs (DFGs)—to identify functional similarity between programs. Their results show that combining heterogeneous edges improves robustness and accuracy. Wang et al. [9] also applied Graph Neural Networks (GNNs) for software defect prediction, demonstrating that structural dependencies captured through graph representations outperform classical ML models that only consider flat metrics.

The growth of publicly available datasets has accelerated research progress. Nguyen et al. [8] compiled and standardized a collection of large-scale vulnerability datasets, enabling model evaluation across diverse security tasks. Complementing this effort, Cheng et al. [4] presented a comprehensive review of AI-driven software vulnerability detection, outlining the evolution from rule-based approaches to neural models and highlighting critical gaps in reproducibility and explainability.

Collectively, these studies reveal three dominant trajectories in code quality research. First, classical ML models using tabular metrics offer ease of interpretation but lack semantic understanding. Second, transformer-based embeddings such as CodeBERT [5] provide rich contextual representations but often operate as black-box systems. Finally, graph-based methods bridge this gap by modeling both semantic and structural dependencies, achieving strong performance while maintaining a degree of interpretability. Despite these advancements, the field still lacks a unified, reproducible, multi-granular framework capable of predicting multiple quality attributes simultaneously—motivating the goals of this work.

2.2 Research Gap

A review of the existing literature indicates several key gaps:

- **Single-dimensional quality focus:** Many studies focus on only one metric—such as maintainability or defectiveness—rather than developing unified, multi-attribute prediction frameworks [4].
- **Limited reproducibility:** Several transformer- and graph-based approaches do not provide end-to-end pipelines, hindering reproducibility and industrial adoption.
- **Insufficient structural modeling in classical ML:** Traditional ML models rely on handcrafted features and cannot capture structural relationships found in ASTs, CFGs, or DFGs [11].
- **Lack of explainability:** While transformer and GNN models achieve strong predictive performance, they remain challenging to interpret, making it difficult to trust their predictions in practical development environments.

2.3 Problem Statement

Modern software systems exhibit significant structural and semantic complexity, making manual or rule-based quality assessment unreliable. While ML and deep learning models have improved code quality prediction, they often focus narrowly on specific metrics or lack reproducibility and interpretability.

This project aims to develop a comparative framework that integrates classical ML, transformer-based models, and graph-based neural networks to predict four key software quality attributes—*Defectiveness, Maintainability, Readability, and Vulnerability.* By leveraging textual, structural, and semantic features in a unified and reproducible pipeline, the framework seeks to provide a comprehensive foundation for automated code quality assessment with improved accuracy and transparency.

Chapter 3

Methodology

This chapter explains the complete methodology followed in Phase I of the project. The main goal of this phase was to build strong text-based baseline models and understand how well traditional methods perform compared to modern transformer-based models. To achieve this, the study focused on two major modeling approaches:

- **Machine Learning models based on TF–IDF representations**, where source code is treated as plain text and converted into numerical features. In most tasks, these TF–IDF features were used to train classical **classification models**. However, for one specific task that required predicting a continuous score, a **regression model** was used instead.

- **A transformer-based approach using the fine-tuned CodeBERT model**, which can learn both syntactic structure and semantic meaning from code, providing deeper and more context-aware representations.

These two approaches were selected to compare traditional feature-based learning with modern deep representation learning. Classical ML methods help establish a simple and interpretable baseline, while CodeBERT offers improved semantic understanding, making it suitable for more complex code-quality prediction tasks.

All four software-quality tasks—Vulnerability, Defectiveness, Maintainability, and Readability—were trained and evaluated using a unified workflow. This ensured that each model received data prepared in the same format and allowed for a fair comparison between methods. The following sections describe each step of the workflow in detail.

3.1 Dataset Preparation

All datasets used in this project were obtained from open-source repositories. To maintain uniformity across tasks, each dataset was converted into the following schema:

```
(code_snippet, label)
```

This ensured that both TF–IDF models and CodeBERT received the same textual input for fair comparison.

- **Vulnerability Dataset:** JISCE CSE AI–ML Vulnerability Fix Dataset.
- **Defectiveness Dataset:** ACM defect dataset used in software defect prediction studies.

- **Maintainability Dataset:** Constructed using maintainability scoring from prior work.
- **Readability Dataset:** Generated using the Buse–Weimer readability scoring tool, with 500 high- and 500 low-readability samples.

Note on dataset sizes and sampling. The four datasets originally contained *different* numbers of total samples. To maintain balanced experimental conditions across tasks and ensure tractable fine-tuning, a subset of the available data was selected for each task. Specifically:

- Vulnerability dataset: used all samples out of the dataset.
- Defectiveness dataset: used **10000** samples out of **116700**.
- Maintainability dataset: used **10000** samples out of **1.4M**.
- Readability dataset: applied the readability tool on the samples from the Maintainability dataset.

These placeholders will be replaced with the exact counts in the final version.

No AST, CFG, DFG, or structural metadata were used in this phase. Models therefore operated purely at the lexical level, processing each snippet independently.

3.2 TF–IDF Based Machine Learning Workflow

A classical machine-learning pipeline was implemented using TF–IDF representations. The complete workflow used in all three classification tasks (and in readability regression) is shown in Figure 3.1.

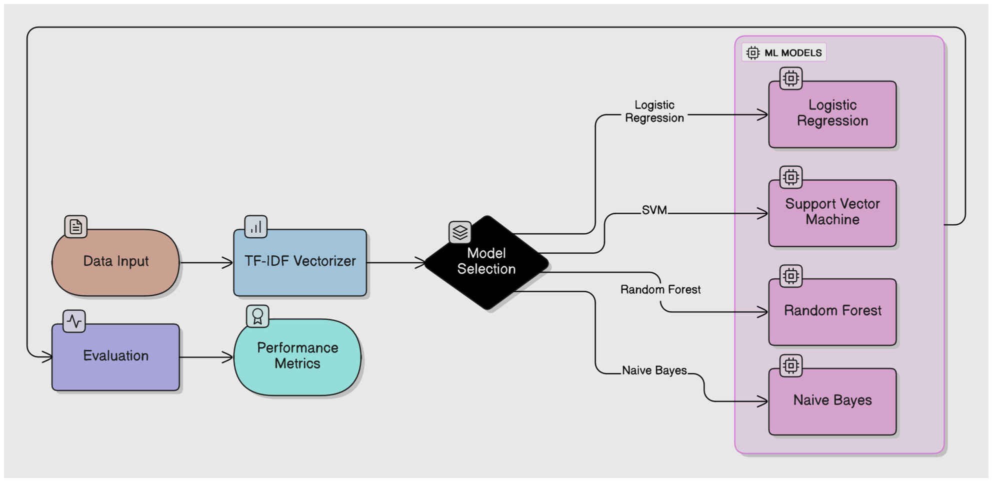

Figure 3.1: Workflow of TF–IDF Vectorization and ML Model Training

Model Selection

For each dataset, the TF–IDF matrix served as input to several classical Machine Learning models. These models were chosen because they are commonly used for text classification and provide strong, interpretable baselines.

- **Logistic Regression:** A linear model used for binary or multi-class classification. It estimates the probability of a class using the logistic (sigmoid) function:

$$P(y = 1 \mid x) = \frac{1}{1 + e^{-w^T x}}$$

Support Vector Machine (SVM): A margin-based classifier that finds a hyperplane separating classes. When using different kernels such as linear or RBF, SVM can capture both simple and non-linear decision boundaries.

Random Forest: An ensemble method consisting of multiple decision trees. The final prediction is obtained by majority voting (classification) or averaging (regression), making it robust to overfitting.

Gradient Boosting: An ensemble technique that builds trees sequentially, where each new tree tries to correct the errors of the previous ones. It often performs better than single decision-tree models due to its boosted learning process.

Each of these models was trained independently for every task to provide a diverse set of baseline comparisons.

Evaluation Metrics

To measure model performance, standard evaluation metrics were used for both classification and regression tasks.

- **Classification Tasks (3 datasets):** The following metrics were used:
  - **Accuracy:**

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

- **Precision:**

$$\text{Precision} = \frac{TP}{TP + FP}$$

- **Recall:**

$$\text{Recall} = \frac{TP}{TP + FN}$$

- **F1-Score:**

$$\text{F1} = 2 \times \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

- **ROC–AUC:** Measures how well the model separates classes across different thresholds. AUC represents the area under the ROC curve.

- **Regression Task (Readability):** The following regression metrics were used:

  - **Mean Absolute Error (MAE):**

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i|$$

  - **Root Mean Squared Error (RMSE):**

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2}$$

  - R^2 **(Coefficient of Determination):**

$$R^2 = 1 - \frac{\sum(y_i - \hat{y}_i)^2}{\sum(y_i - \bar{y})^2}$$

Measures how well the regression model fits the data.

These metrics provide a reliable baseline to understand how lexical, bag-of-words style representations using TF–IDF perform across both classification and regression tasks.

3.3 CodeBERT Fine-Tuning Workflow

Transformer-based modeling was implemented using the `microsoft/codebert-base` encoder. Figure 3.2 shows the exact architecture used for all tasks.

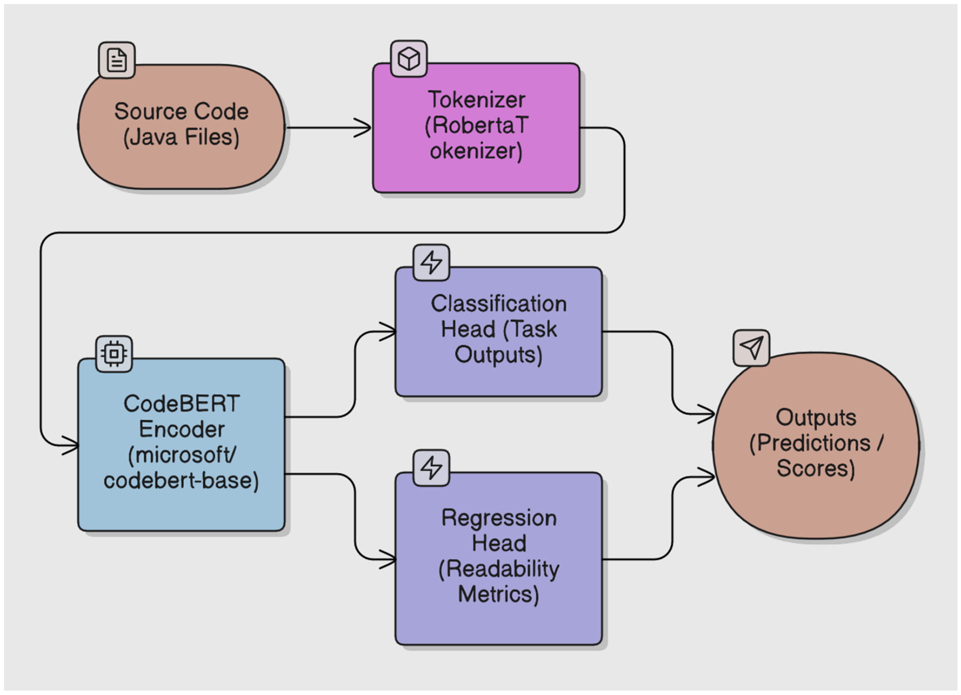
Source Code (Java Files)
Tokenizer (RobertaT okenizer)
CodeBERT Encoder (microsoft/ codebert-base)
Classification Head (Task Outputs)
Regression Head (Readability Metrics)
Outputs (Predictions / Scores)

Figure 3.2: Workflow of CodeBERT Fine-Tuning for Classification and Regression Tasks

Tokenization

Source code (Java) was tokenized using the RobertaTokenizer, producing:

`input_ids`, `attention_mask`

Encoder and Task-Specific Heads

- The **CodeBERT encoder** produced contextual embeddings.
- A **classification head** was used for Vulnerability, Defectiveness, Maintainability.
- A **regression head** was used for Readability scoring.

Training

The transformer model (CodeBERT) was fine-tuned separately for each task. Fine-tuning means updating the pretrained model weights on our specific datasets so that the model learns task-specific patterns from the source code. All tasks were trained for a total of 3 epochs, which was sufficient to observe convergence without overfitting.

The following components were used during training:

- **AdamW optimizer:** A variant of Adam that includes proper weight decay. It helps maintain stable updates and prevents the model from overfitting by controlling the magnitude of the weights.
- **Linear learning-rate decay:** The learning rate starts with a small initial value and gradually decreases to zero. This helps the model

make large updates early on and smaller, more precise updates toward the end of training.

- **Cross-entropy loss (for classification tasks):** This loss function measures how far the predicted probability distribution is from the true class. It is defined as:

  $$\mathcal{L}_{CE} = -\sum_{i=1}^{n} y_i \log(\hat{y}_i)$$

  where y_i is the true label and $\hat{y}_i$ is the predicted probability.

- **Mean Squared Error (MSE) loss (for regression task):** Used for predicting continuous readability scores. It is defined as:

  $$\mathcal{L}_{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Together, these components created a strong transformer-based baseline for comparing with the TF–IDF Machine Learning models.

3.4 Evaluation Framework

To ensure a fair and consistent comparison between models, every approach (TF–IDF ML models and CodeBERT) was evaluated using the same framework. The goal of this evaluation process is to understand not only whether a model is accurate, but also how and where it makes mistakes.

The following visualizations were generated for each dataset:

- **Confusion Matrices (all classification tasks):** These matrices show how often the model correctly predicts each class versus how often it confuses it with another class. They help identify patterns such as class imbalance or systematic misclassification.

- **ROC–AUC Curves (for ML classification models):** The ROC curve plots True Positive Rate vs. False Positive Rate across different thresholds. The AUC value summarizes this curve into a single score, indicating how well the model separates positive and negative classes.

- **Training-loss Curves (for CodeBERT):** These curves show how the loss decreases over time during training. A smooth downward trend indicates stable learning, while fluctuations may suggest instability or overfitting.

- **Predicted vs. Actual Plots (Readability regression task):** These plots compare the model's predicted readability scores against the true values. Points close to the diagonal line represent accurate predictions, while scattered points indicate larger errors.

This evaluation framework ensures that both traditional machine learning models and transformer-based models can be compared objectively, highlighting their strengths and limitations across different code-quality prediction tasks.

Chapter 4

Results and Discussion

This chapter presents a consolidated discussion of all experimental findings from Phase I. Since performance metrics for all four datasets were already reported in the previous chapter, this section focuses exclusively on the visual analysis using the collected plots — confusion matrices, ROC curves, regression fits, and training curves. The goal here is not to restate numerical values, but to interpret model behavior using visual evidence and understand the strengths and limitations of each modeling approach.

4.1 Vulnerability Prediction

The vulnerability dataset exhibited strong lexical separability, and both TF–IDF based Machine Learning models and the CodeBERT model performed competitively. Most vulnerable code samples contained common patterns and specific API calls, which allowed TF–IDF models such as Logistic Regression and SVM to classify them effectively. These models performed well because many vulnerabilities depend on repeated keywords that TF–IDF representations can easily capture.

CodeBERT also achieved strong performance, especially on cases where the vulnerability was more semantic and depended on the overall control flow rather than isolated tokens. Its confusion matrix showed fewer misclassifications in examples where similar lexical patterns belonged to different classes. This indicates that CodeBERT benefits from understanding context and structure, enabling it to resolve ambiguous samples more accurately.

Overall, the visual analysis shows that although simple lexical models provide strong baselines for vulnerability detection, CodeBERT adds value by handling harder, context-dependent examples with greater consistency.

4.1.1 Performance Summary

Model	**Accuracy**	**F1-Score**
Logistic Classifier	96%	96%
SVM (RBF)	96%	96%
Random Forest	96.06%	96.02%
Gradient Boosting	96.03%	96.01%
Fine-Tuned CodeBERT	**99.96%**	**99.92%**

Table 4.1: Model Performance on the Vulnerability Dataset

4.1.2 Confusion Matrices of ML Models

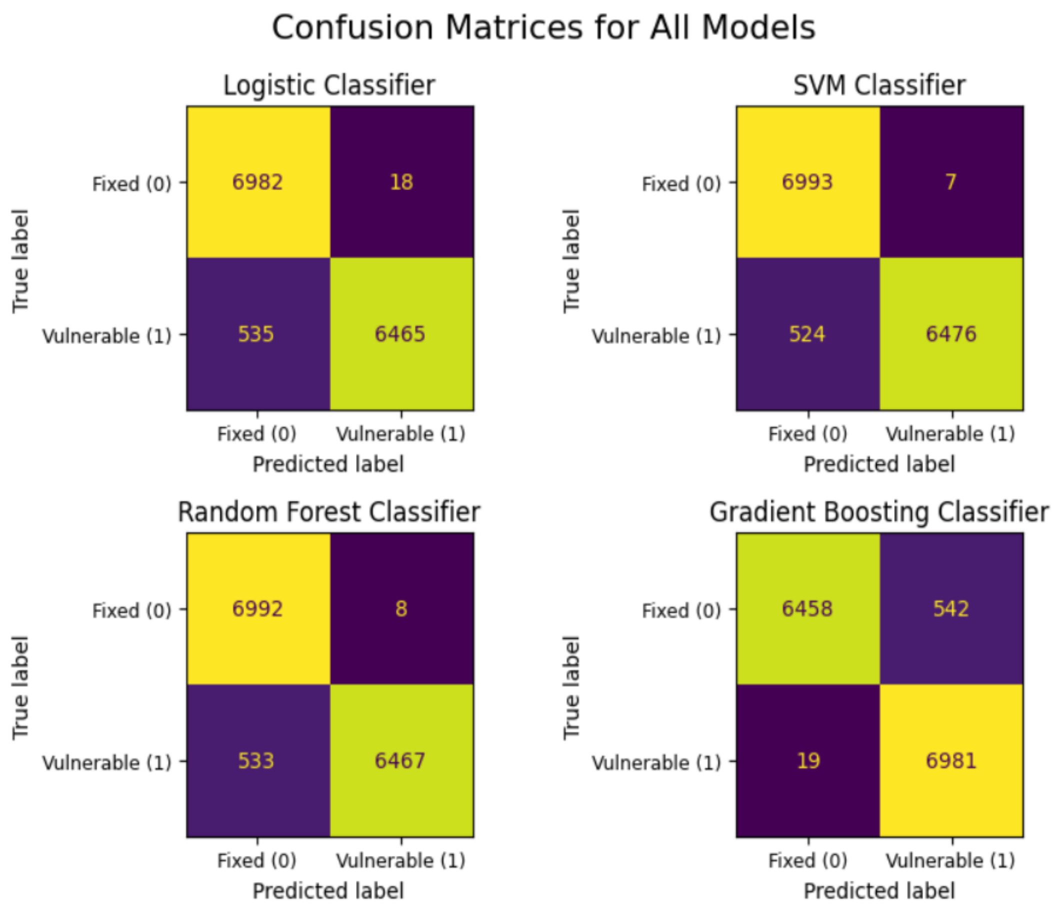

Figure 4.1: Confusion Matrices of Logistic Classifier, SVM (RBF), Random Forest, and Gradient Boosting.

The majority of models show very few misclassifications, indicating that vulnerability patterns contain discriminative lexical signals such as missing validation checks and insecure API usage.

4.1.3 ROC–AUC Curves of ML Models

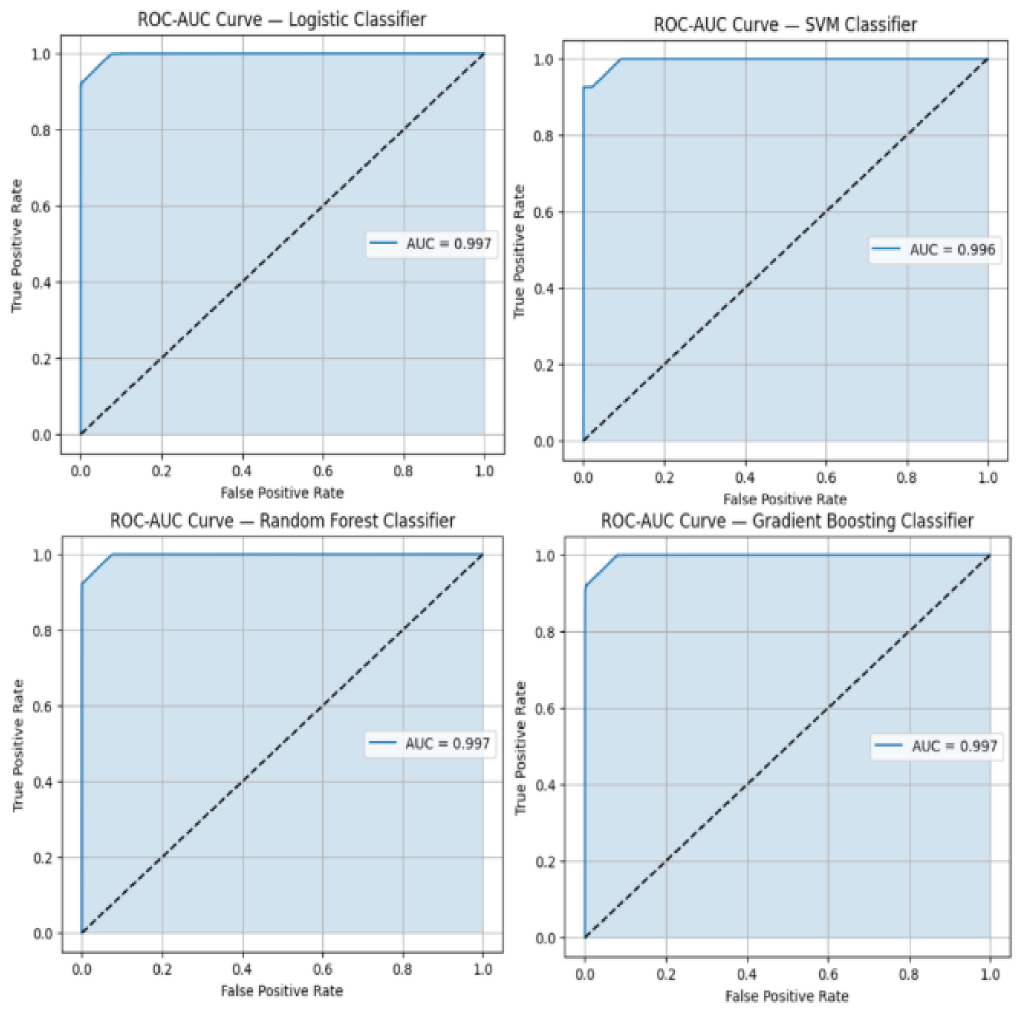

Figure 4.2: ROC–AUC Curves of ML Models on the Vulnerability Dataset.

4.1.4 CodeBERT Confusion Matrix

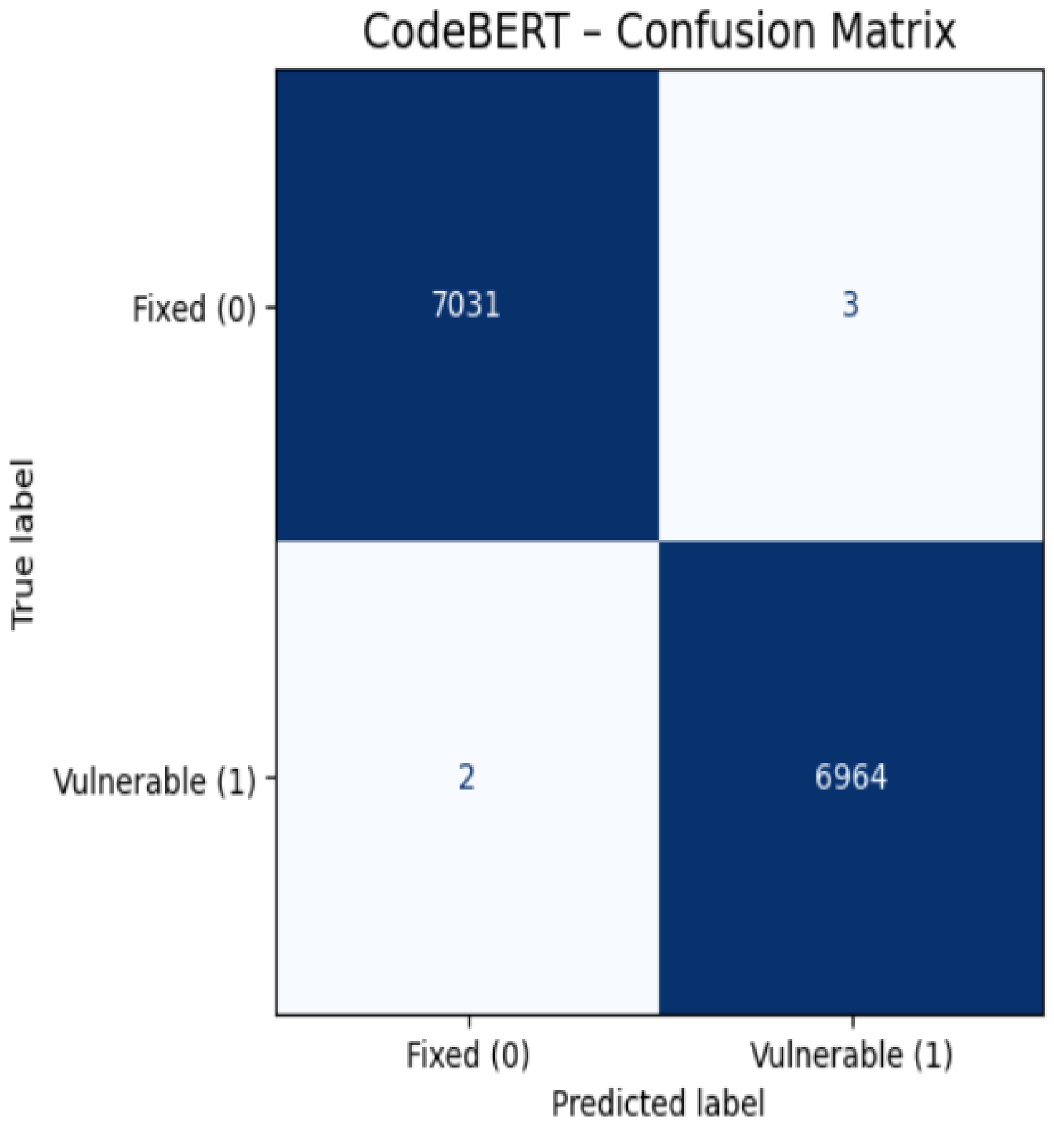

Figure 4.3: Confusion matrix of the fine-tuned CodeBERT vulnerability classifier.

4.1.5 CodeBERT ROC–AUC Curve

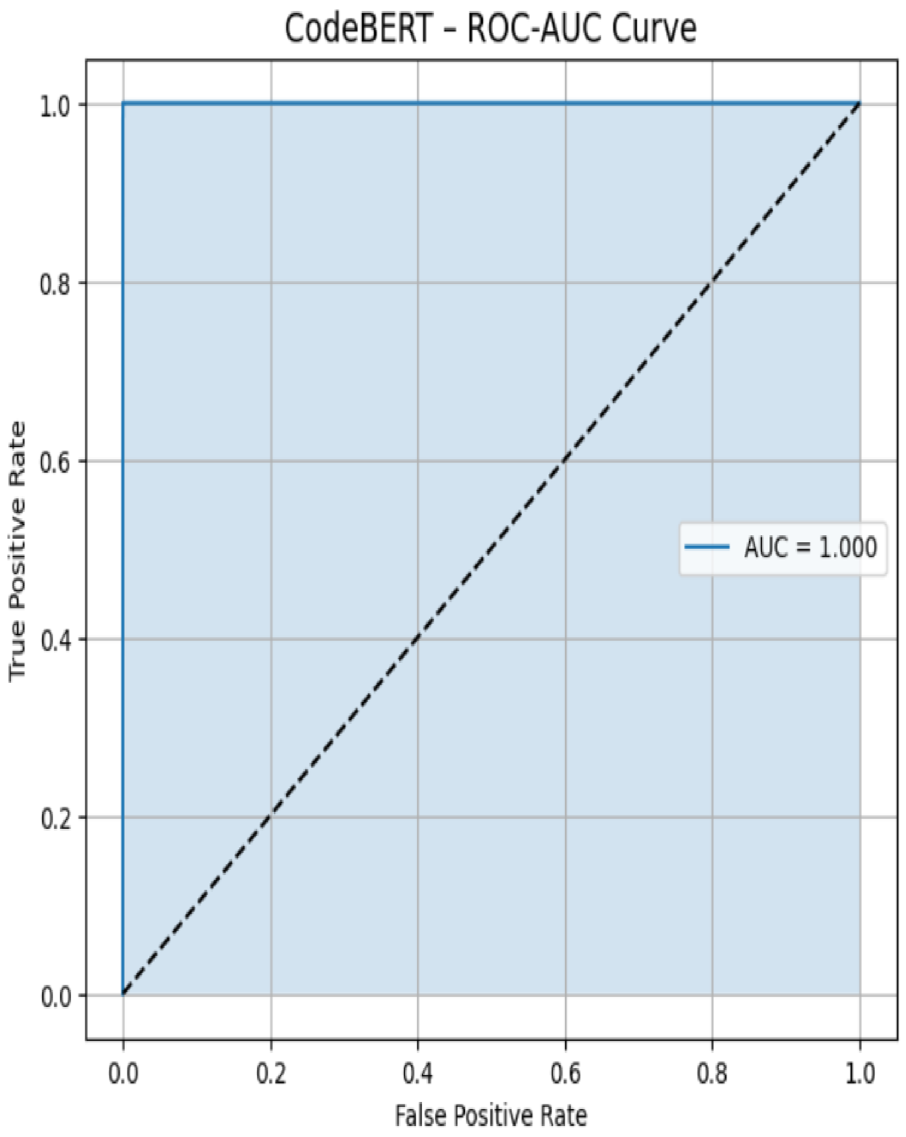

Figure 4.4: ROC–AUC curve of the fine-tuned CodeBERT model on the vulnerability dataset.

4.1.6 Training Loss Curve of CodeBERT

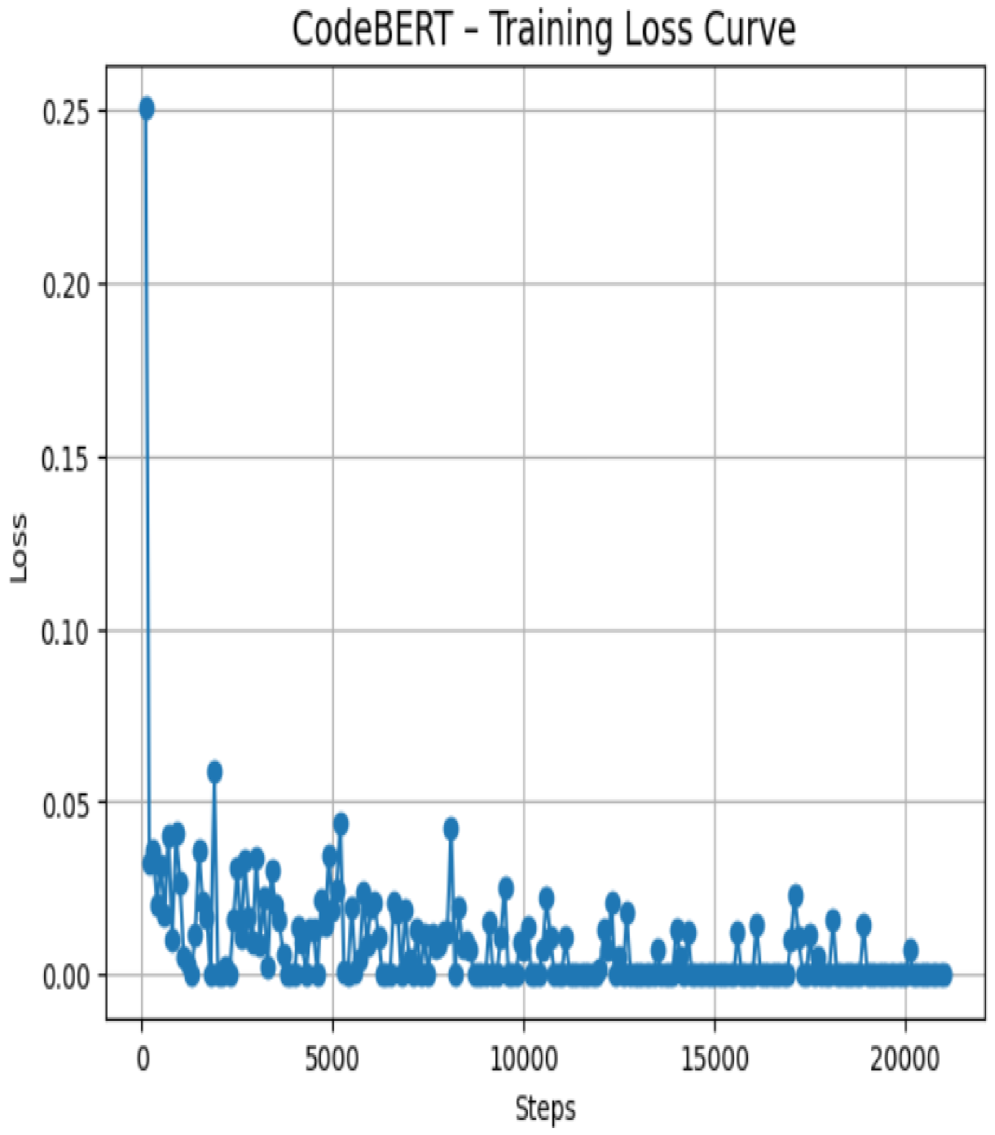

Figure 4.5: Training loss curve of the fine-tuned CodeBERT model across epochs.

4.2 Defectiveness Prediction

4.2.1 Performance Summary

Model	**Accuracy**	**F1-Score**
SGD Logistic Classifier	51.70%	51.68%
SVM (Linear)	51.40%	51.40%
Random Forest	51.65%	51.51%
Gradient Boosting	51.65%	51.51%
Fine-Tuned CodeBERT	57.55%	56.96%

Table 4.2: Model Performance on the Defectiveness Dataset

The defectiveness dataset was the most challenging due to its weak information per learning sample. Most code snippets were short and contained very few meaningful tokens, which made it harder for TF–IDF models to extract strong lexical patterns. As a result, the confusion matrices showed more overlap between defective and non-defective classes, indicating that the model often struggled to find clear boundaries.

CodeBERT performed slightly better, but even the transformer model faced difficulty because many defects depend on deeper structural or logical issues that are not visible in small code fragments. This limited the amount of useful context available to both lexical and semantic models.

Overall, the visual analysis suggests that this dataset would benefit from richer contextual information or larger code segments, as the current samples do not contain enough signals for robust defect prediction.

4.2.2 Confusion Matrices of ML Models

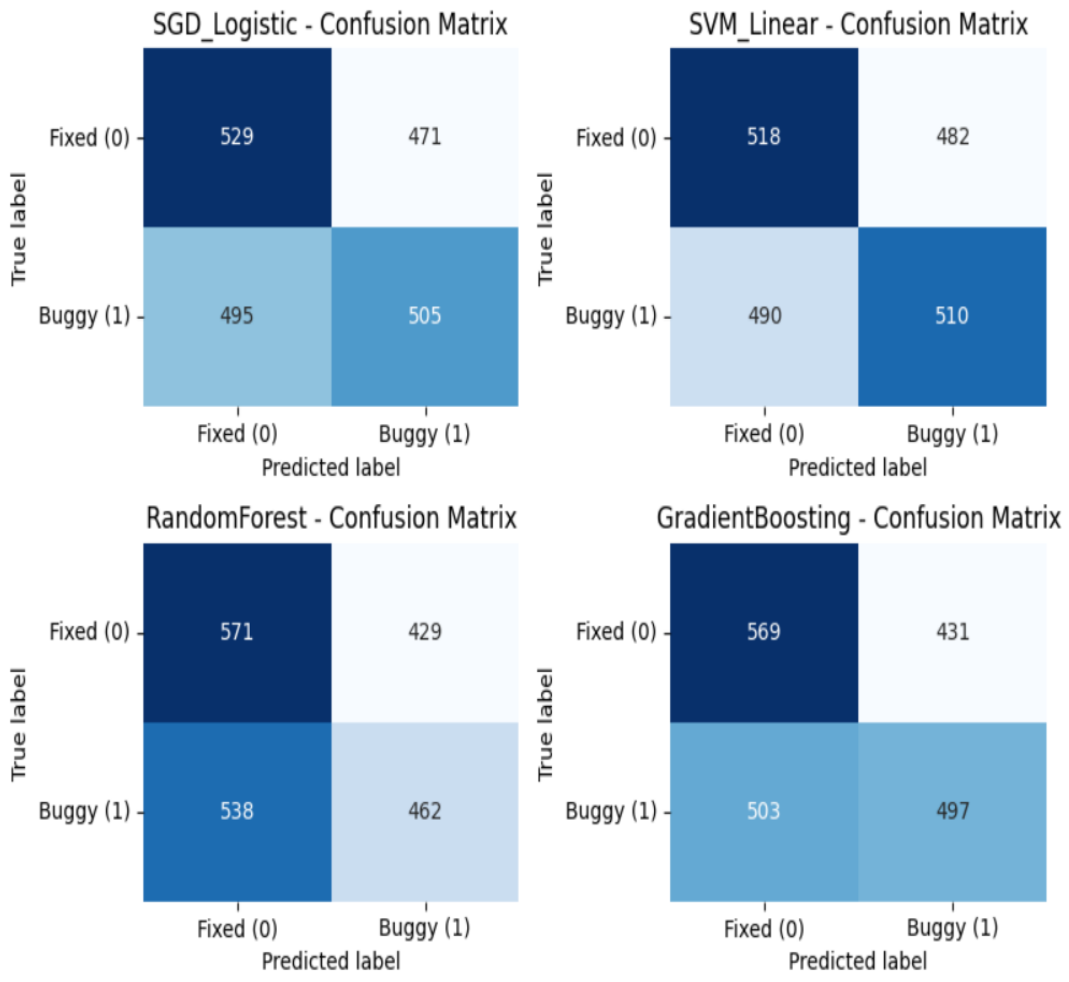

Figure 4.6: Confusion matrices of ML models on the defectiveness dataset.

4.2.3 ROC–AUC Curves of ML Models

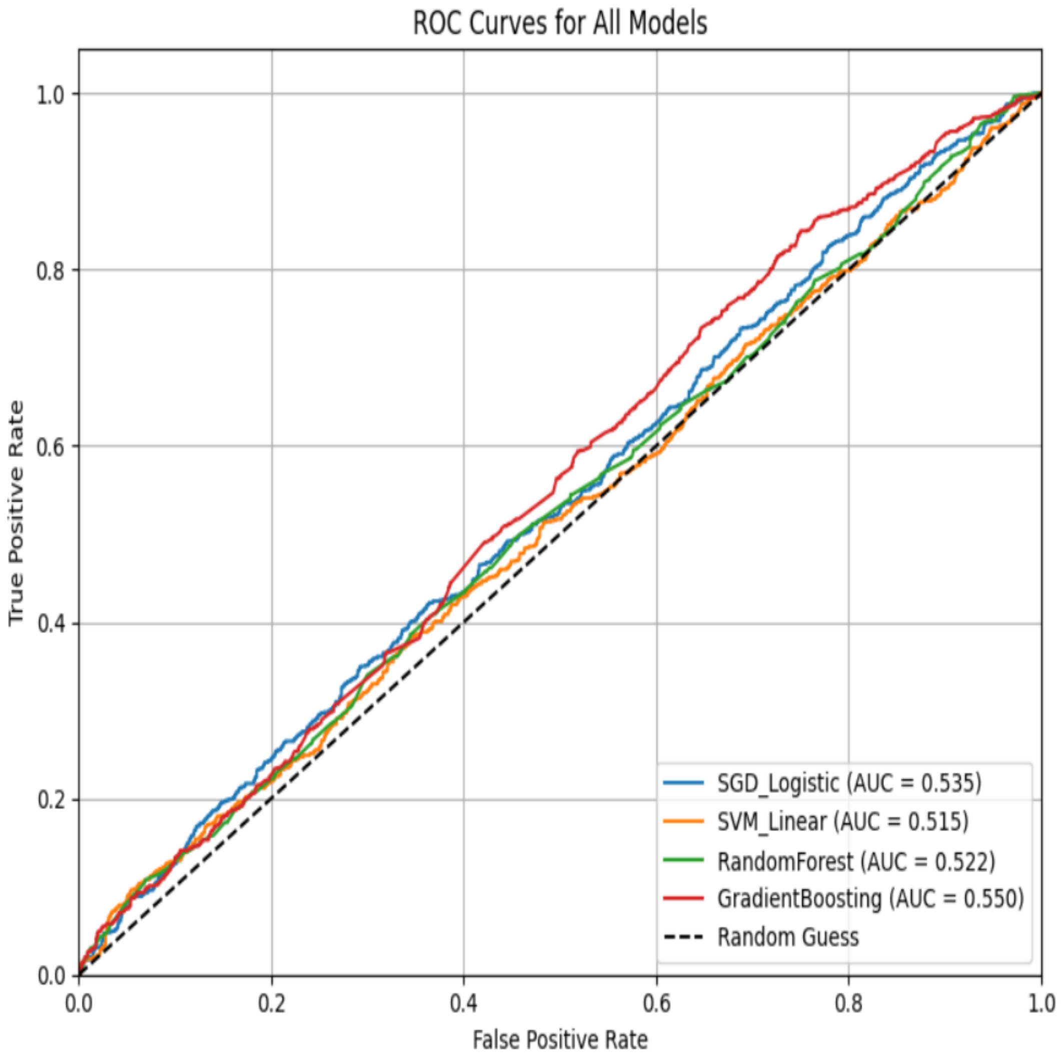

Figure 4.7: ROC–AUC curves of ML models on the defectiveness dataset.

4.2.4 CodeBERT Confusion Matrix

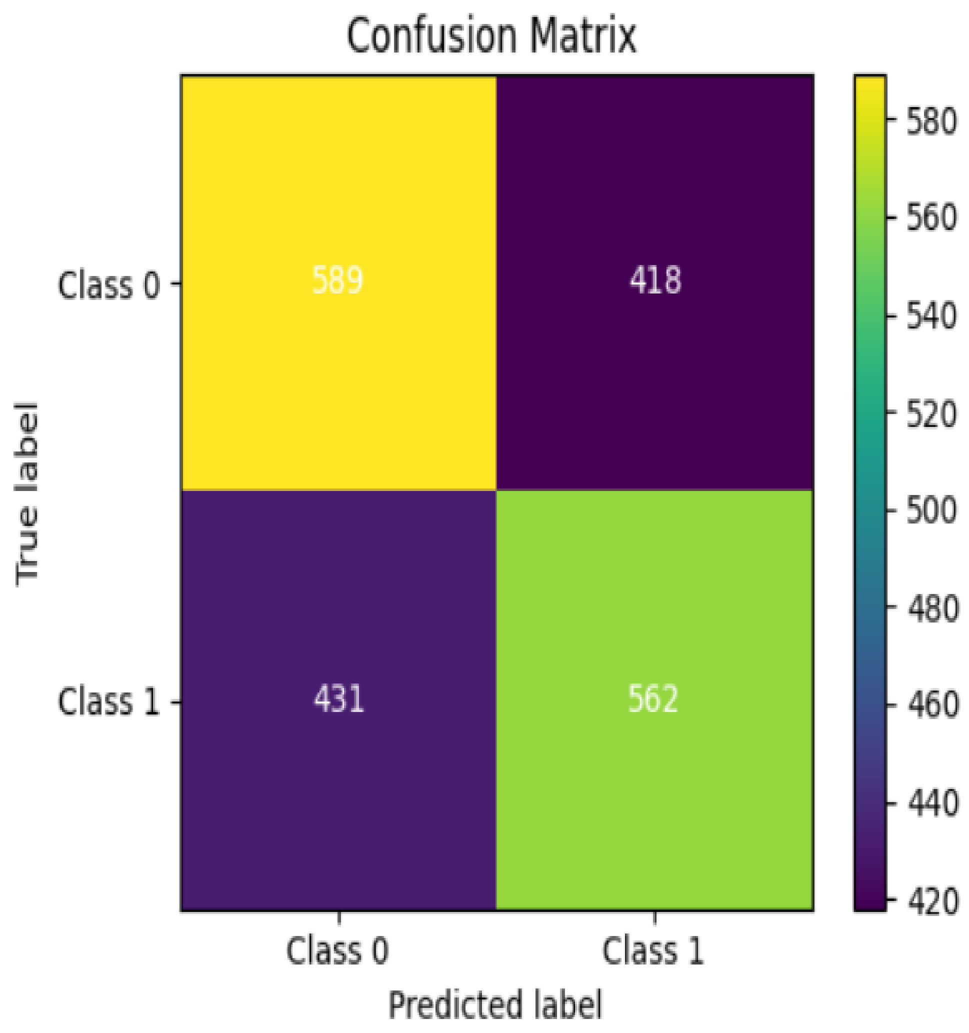

Figure 4.8: Confusion matrix of the fine-tuned CodeBERT defectiveness classifier.

4.2.5 CodeBERT ROC Curve

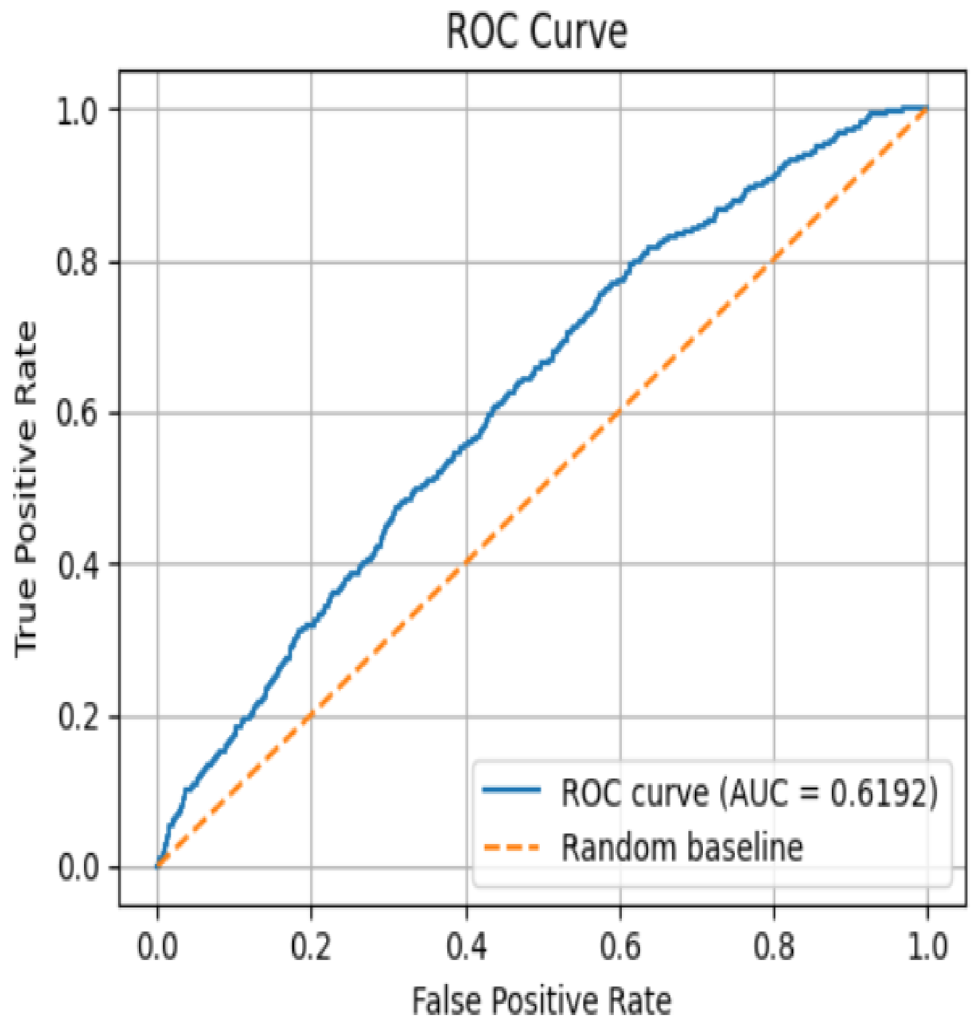

Figure 4.9: ROC curve of the fine-tuned CodeBERT model on the defectiveness dataset.

4.2.6 Training Loss Curve of CodeBERT

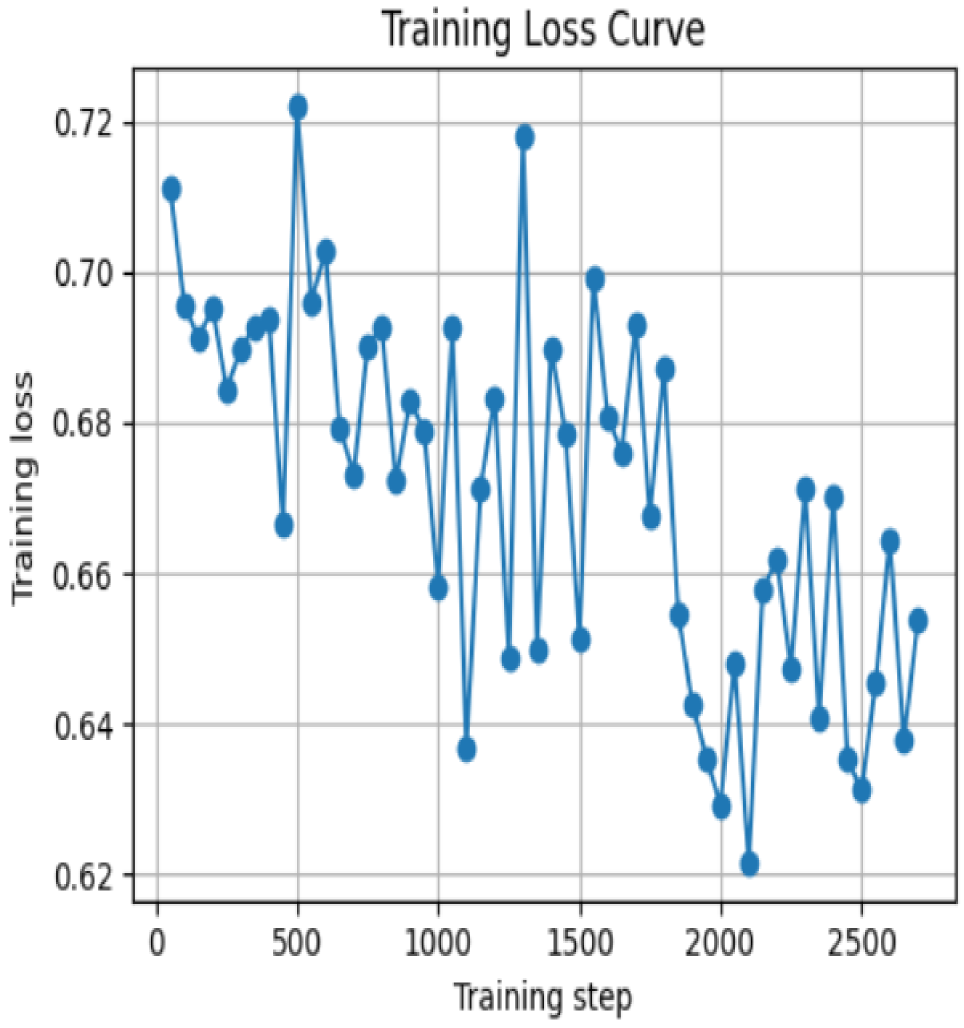

Figure 4.10: Training loss curve of the fine-tuned CodeBERT model across epochs for defectiveness prediction.

4.3 Maintainability Prediction

4.3.1 Performance Summary

Model	**Accuracy**	**F1-Score**
Logistic Regression	93.75%	0.94
SVM (Linear)	93.70%	0.94
Random Forest	94.70%	0.95
Gradient Boosting	94.90%	0.95
Fine-Tuned Code-BERT	93.25%	0.936

Table 4.3: Model Performance on the Maintainability Dataset

Maintainability demonstrated strong lexical biases, leading to consistently high ML performance.

4.3.2 Confusion Matrices of ML Models

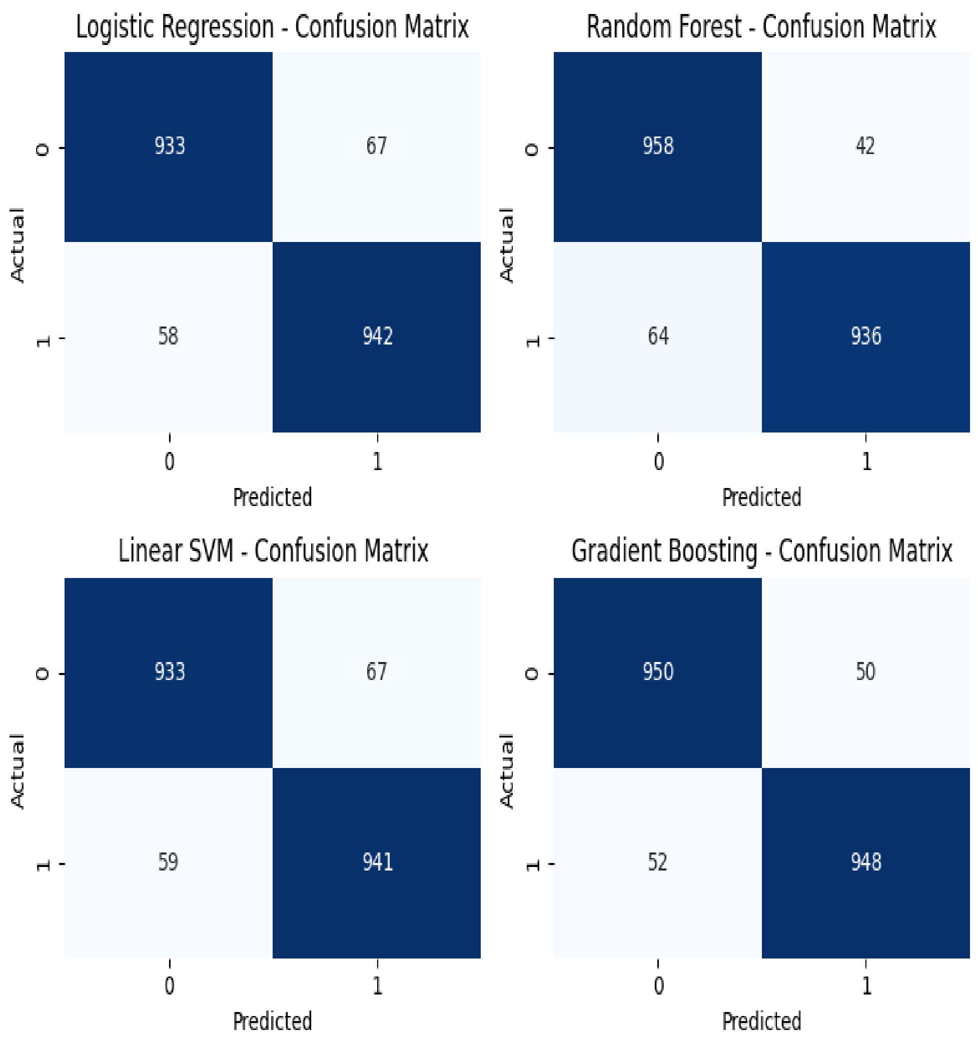

Figure 4.11: Confusion Matrices of Logistic Regression, SVM, Random Forest, and Gradient Boosting

4.3.3 ROC-AUC Curves of ML Models

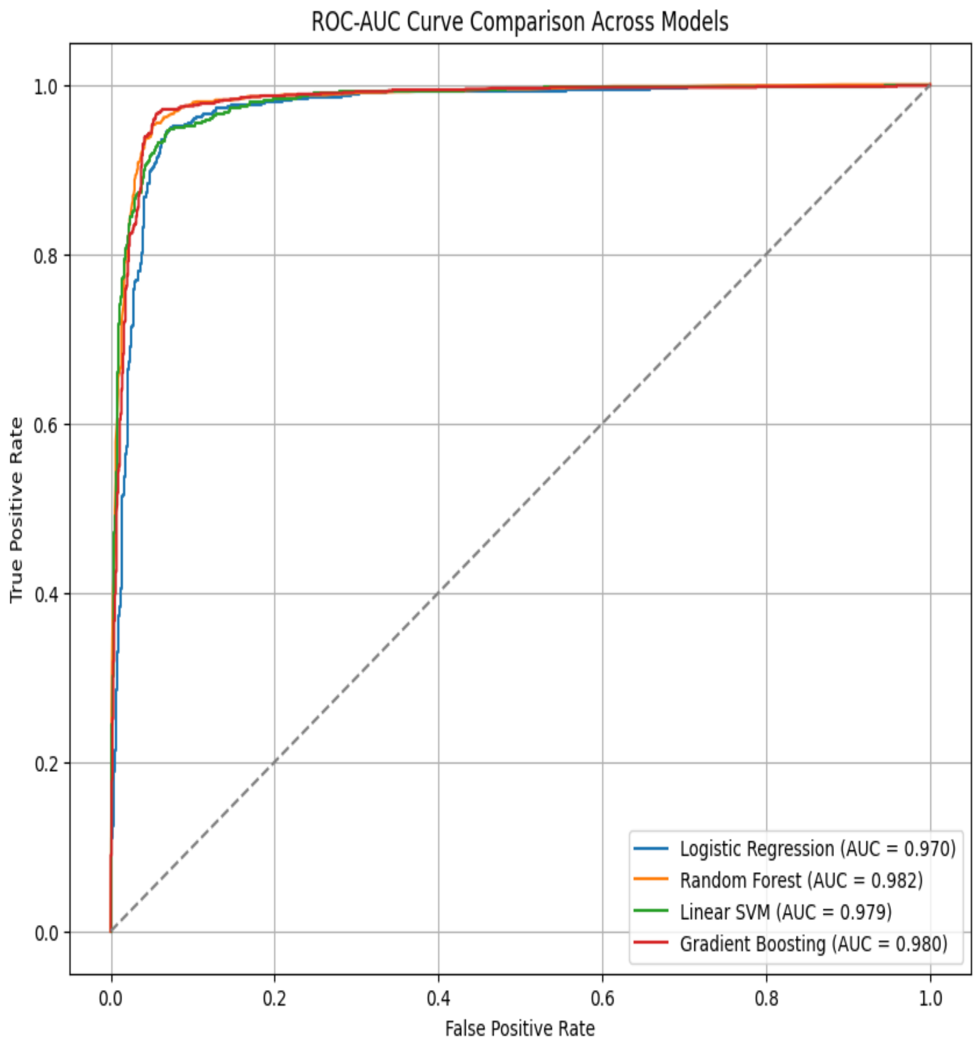

Figure 4.12: ROC-AUC Curves of ML Models on maintainability dataset.

4.3.4 CodeBERT Training Curve and Confusion Matrix

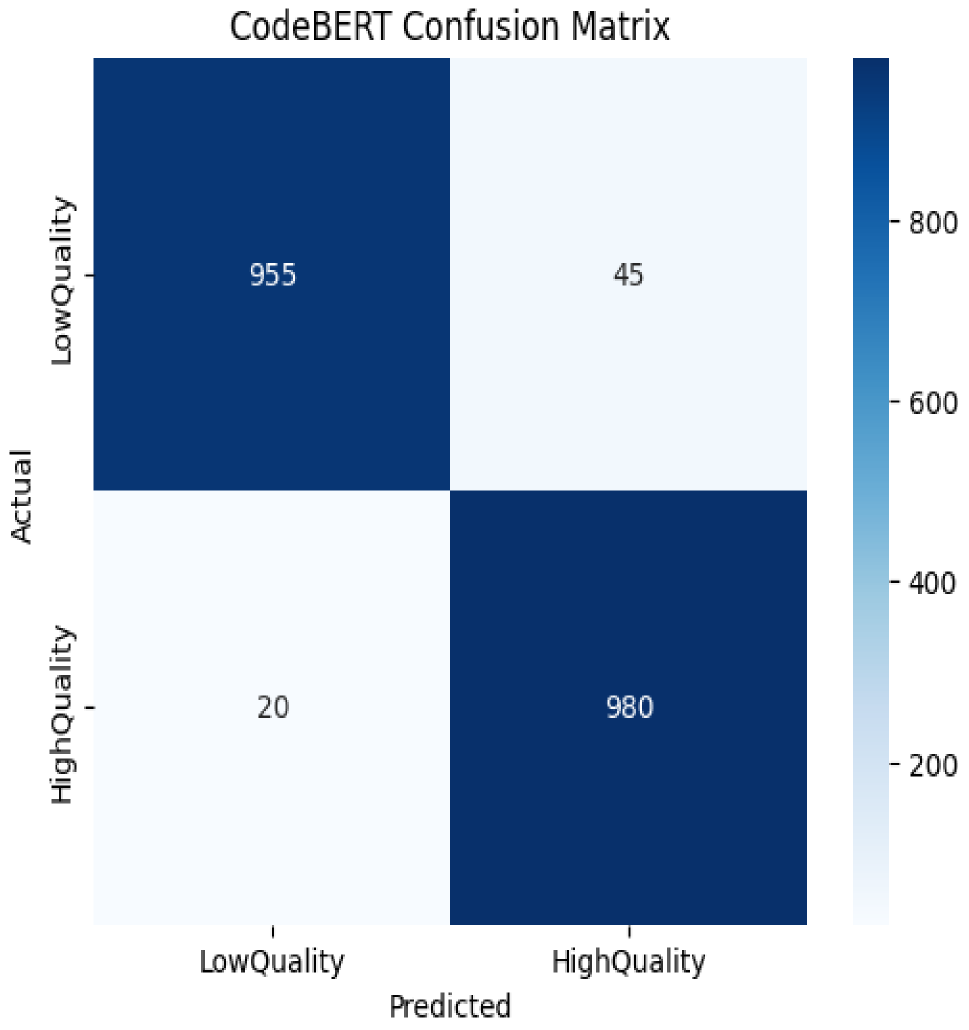

Figure 4.13: Confusion Matrix of Fine-Tuned CodeBERT for Maintainability

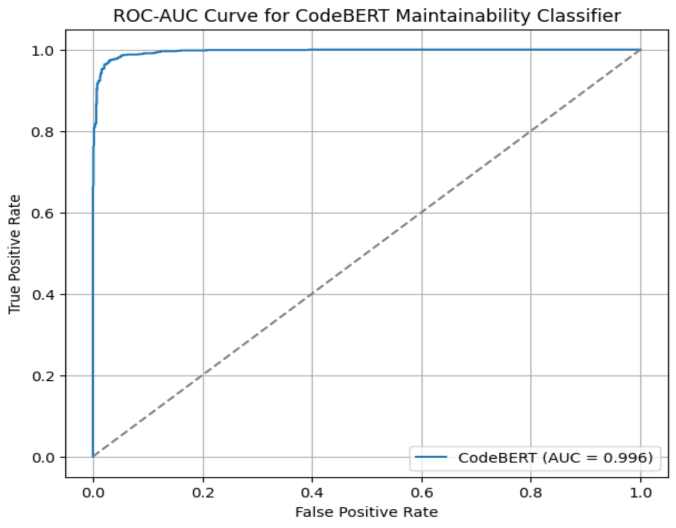

Figure 4.14: Training loss curve of the fine-tuned CodeBERT model across epochs for Maintainability

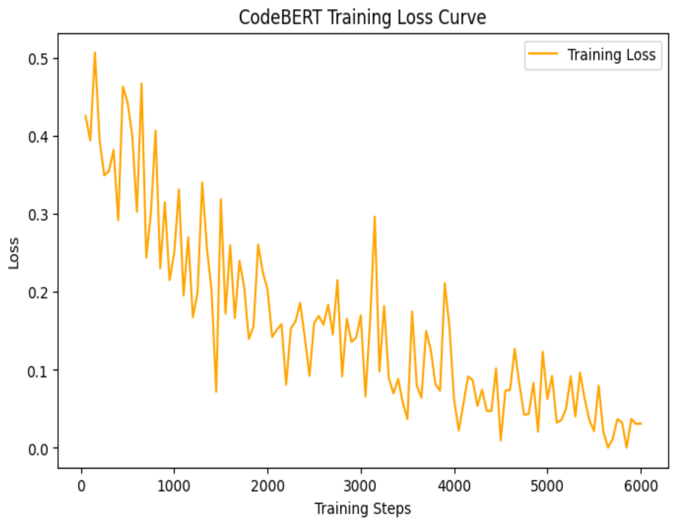

Figure 4.15: CodeBERT Training Loss Curve for Maintainability

4.4 Readability Prediction

4.4.1 Performance Summary

Model	**MAE**	**RMSE**	**R^2**
Linear Regression	0.1269	0.1771	0.2760
Support Vector Regressor	0.1251	0.1598	0.4106
Random Forest Regressor	**0.1031**	**0.1486**	**0.4904**
Gradient Boosting Regressor	0.1065	0.1517	0.4691
Fine-Tuned CodeBERT	0.1191	0.1690	0.3411

Table 4.4: Model Performance on the Readability Dataset

4.4.2 Performance Comparison of ML Models and CodeBERT

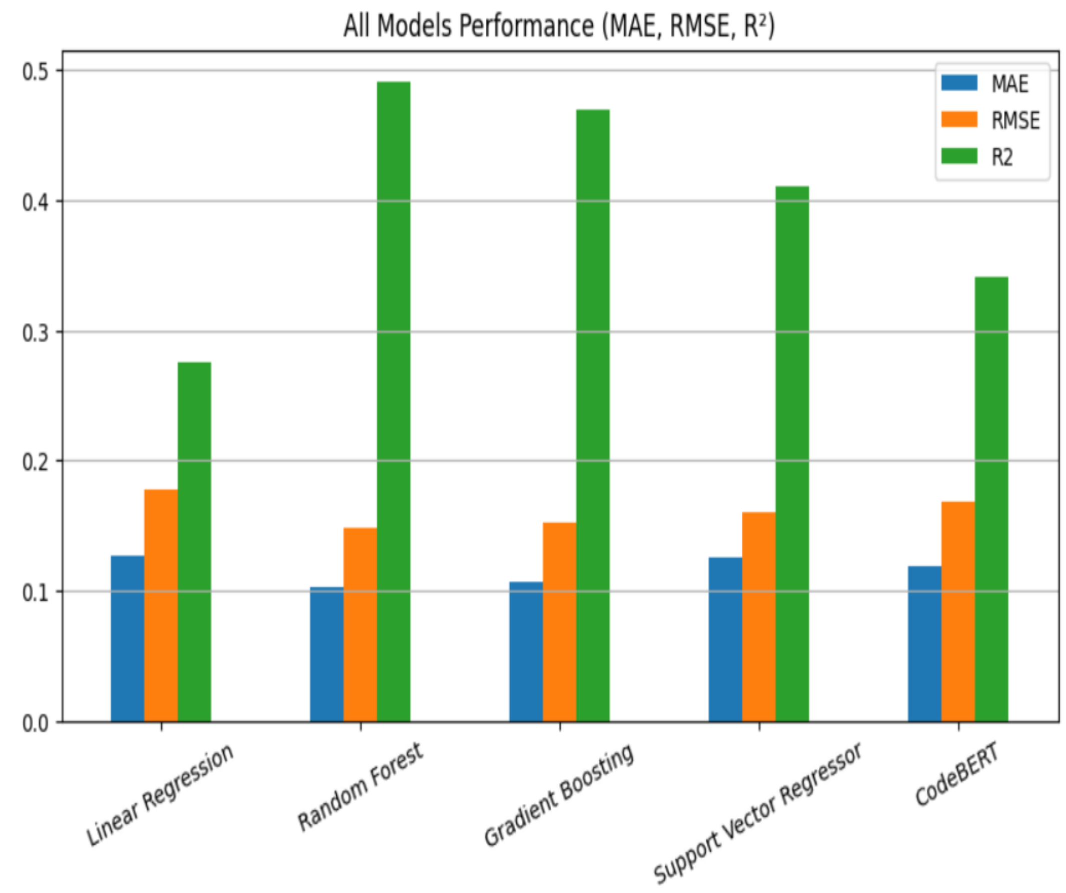

Figure 4.16: MAE, RMSE, and R^2 Comparison Across ML Models and CodeBERT

4.4.3 CodeBERT Regression Fit

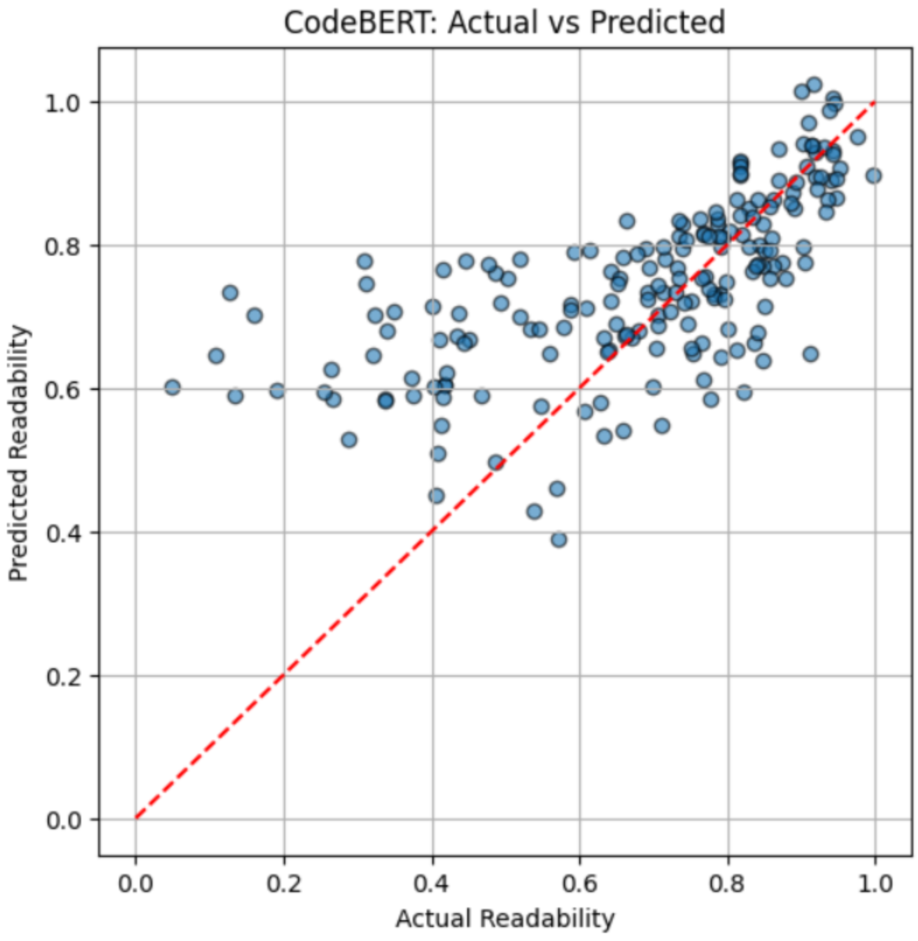

Figure 4.17: CodeBERT Predicted vs Actual Readability Scores (Line Fit)

4.5 Overall Discussion

Across all datasets, we observe consistent behavioral patterns:

- **Vulnerability:** Both ML and CodeBERT perform strongly, with Code-

BERT achieving near-perfect accuracy due to semantic understanding of insecure patterns.

- **Defectiveness:** All models struggle due to weakly separable labels and limited lexical cues.
- **Maintainability:** TF-IDF–based ML models perform best because the labels correlate heavily with surface-level code patterns.
- **Readability:** Ensemble regressors outperform CodeBERT owing to the dataset's small size and syntactic nature of readability features.

These insights justify the planned Phase II extension toward structure-aware Graph Neural Networks, which can potentially capture relationships that TF-IDF and CodeBERT alone cannot represent.

Chapter 5

Conclusion and Future Work

5.1 Limitations of Phase I Models: Lack of Granularity Awareness

An important limitation of the models developed in this phase is that none of them are **granularity-aware**. All experiments were conducted on datasets structured strictly as `(code, label)` pairs, where each sample represented an isolated snippet without explicit information about:

- the function it belongs to,
- the surrounding file or module,
- control-flow or data-flow relationships,
- or its role within a larger software component.

As a result, both TF–IDF based ML models and the fine-tuned CodeBERT model operated only at a **flat, snippet-level perspective**. This restriction affects tasks such as defectiveness or maintainability, where quality often depends on structural and contextual cues that extend beyond the snippet itself.

Why TF–IDF based ML models performed unusually well on small datasets

The strong performance of TF–IDF models on vulnerability and maintainability tasks can be attributed to three factors:

1. **Token-level patterns dominate the labels.** Many of the labels (e.g., maintainability heuristics, vulnerability signatures) correlate strongly with surface-level lexical features such as keyword frequency, identifier length, or presence/absence of validation logic.

2. **Small datasets favor simpler hypothesis spaces.** Classical models such as Logistic Regression, SVM, Random Forest, and Gradient Boosting are able to learn stable decision boundaries from limited data. By contrast, CodeBERT—with over 100M learnable parameters—is far more sensitive to dataset size and noise.

3. **Absence of structural granularity reduces the advantage of semantic models.** CodeBERT can model deeper semantics, but when datasets do not contain structural or multi-level information, its advantage diminishes. TF–IDF models succeed because they align directly with lexical regularities that strongly correlate with the target labels.

5.2 Fallbacks of ignoring code granularity

Because all inputs were treated as independent text blocks, the models lack awareness of:

- inter-function dependencies,
- project-level organization,
- module cohesion or coupling,
- control-flow properties affecting defectiveness,
- and readability signals that arise from structural layout.

This limitation is especially visible in the defectiveness and readability tasks, where the snippet-level representation provided insufficient discriminatory power for reliable classification or regression.

Motivation for Phase II

These insights make it clear that future models must incorporate **multi-granular** information and **multi-task learning**. Structure-aware approaches—such as GNNs trained on ASTs, CFGs, and DFGs—have the potential to capture semantic and relational properties that flat text models inherently miss.

This realization directly motivated the design goals for Phase II.

5.3 Conclusion

This phase of the project presented a systematic comparison between traditional TF-IDF based machine learning models and a fine-tuned CodeBERT model across four dimensions of code quality: Vulnerability, Defectiveness, Maintainability, and Readability. The study standardized all datasets into a single (code, label) schema and applied a consistent preprocessing and training pipeline for both ML and transformer-based models.

The results highlight several important patterns. Models trained on TF-IDF features performed exceptionally well on tasks where lexical cues dominate, such as maintainability and vulnerability prediction. Ensemble models in particular demonstrated strong generalization due to their ability to exploit surface-level token distributions present in source code. On the other hand, CodeBERT showed clear advantages in scenarios where deeper semantic relationships influenced the labels—most notably in vulnerability prediction, where contextual understanding of insecure programming patterns provided a significant boost in accuracy.

Defectiveness prediction emerged as the most challenging task for all approaches, revealing the limited discriminative power of both lexical and semantic signals in the available dataset. For readability, the classical regressors outperformed CodeBERT, largely due to the small dataset size and the nature of readability scores, which correlate more with stylistic traits than with semantic behavior.

Overall, the findings of this phase establish strong TF-IDF baselines while also revealing the potential and limitations of transformer-based models in code quality prediction. These insights form a grounded foundation for more advanced modeling strategies in the next phase of the project.

5.4 Future Work

The next phase of this project will extend beyond text-only modeling and move toward structure-aware approaches by training Graph Neural Network (GNN) models on code representations such as Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), and Data Flow Graphs (DFGs). A core objective of this phase is to design and train **multi-task, multi-granular GNN architectures** capable of learning several code quality dimensions simultaneously while operating at different levels of abstraction.

These upcoming models will be explicitly constructed to:

- perform **multi-task learning**, jointly predicting defectiveness, maintainability, readability, and vulnerability using a shared GNN encoder with separate heads for each task;
- leverage **structural information** from AST, CFG, and DFG edges to capture code semantics that TF-IDF and CodeBERT cannot fully represent.

In addition to structural modeling, the next phase will possibly explore **explainability mechanisms** to understand why a model arrives at a particular prediction. Future work will investigate:

- how specific AST nodes or control-flow edges influence a GNN's multi-task prediction,
- how attention patterns in transformers and GNNs highlight critical regions of code,

- how explanations differ between ML, CodeBERT, and GNN-based models.

These planned developments will allow the system to evolve from single-task lexical prediction into a unified, structure-aware, multi-task, and multi-granular code quality assessment framework with interpretable outputs.

Bibliography

[1] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. In *Proceedings of the ACM on Programming Languages*, volume 3, pages 1–29. ACM, 2019.

[2] Lionel C. Briand, John W. Daly, and Jürgen K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999.

[3] Raymond P. L. Buse and Westley R. Weimer. Learning a metric for code readability. In *IEEE Transactions on Software Engineering*, volume 36, pages 546–558. IEEE, 2010.

[4] Xiang Cheng, Hui Wang, Yang Liu, and Tianwei Zhang. Ai-driven software vulnerability detection: A comprehensive survey. *IEEE Transactions on Software Engineering*, 50(2):245–270, 2024.

[5] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages.

In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, 2020.

[6] Lalit Kumar and Ashish Sureka. Deep learning-based software maintainability prediction. *Journal of Systems and Software*, 185:111161, 2022.

[7] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, (4):308–320, 1976.

[8] Van Nguyen, Jianguo Chen, and Wei Zhang. Vulnerability datasets for software security: A comprehensive survey. *ACM Computing Surveys*, 56(3):1–38, 2024.

[9] Song Wang, Taiyue Liu, and Lin Tan. Software defect prediction based on graph neural networks. *Empirical Software Engineering*, 27(4):1–33, 2022.

[10] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 87–98. IEEE, 2016.

[11] Alexey Zeng, Miltiadis Allamanis, Marc Brockschmidt, Alexander L. Gaunt, and Oleksandr Polozov. Tailor: Generating and perturbing text with semantic controls. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 3194–3213, 2021.