Semantic and Structural Fusion for Code Smell Detection Using CodeBERT and Random Forest

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Sohail Shaik

Dec 2025

The Designated Project Committee Approves the Project Titled

Semantic and Structural Fusion for Code Smell Detection Using CodeBERT and Random Forest

by

Sohail Shaik

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

Dec 2025

Dr. Robert Chun	Department of Computer Science
Dr. Sayma Akther	Department of Computer Science
Nava Prashanth Kakarla	Software Engineer

ABSTRACT

Semantic and Structural Fusion for Code Smell Detection Using CodeBERT and Random Forest

by Sohail Shaik

This project explores automatic code smell detection in Java code using transformer-based embeddings and machine learning classifiers. I utilize Code-BERT, a pre-trained transformer model, to extract semantic features from code snippets and evaluate the efficacy of Random Forest and Neural Network classifiers. The study uses the SmellyCode++ dataset, which has 107,554 Java code examples with four types of code smells: Long Method, God Class, Feature Envy, and Data Class. My methodology comprises extracting 768-dimensional embeddings with CodeBERT, training two separate classifiers, and assessing their performance on a balanced subset of 5,000 samples. Presented the metrics of accuracy, precision, recall, F1-score, and confusion matrices, and verify generalization through the application of real-world Java examples. Class imbalance is addressed through stratified sampling and the implementation of a balanced training division. Both models performed similarly with an accuracy of around 78The results show that CodeBERT embeddings are good at finding semantic patterns in code structure. This makes it possible to find code smells and gives us a standard that I can use for future multi-label extensions and large-scale deployment. Beyond metrics, I recorded preprocessing decisions, hyperparameter configurations, and inference procedures to ensure complete reproducibility, and I address potential threats to validity. I delineated integration pathways into IDEs and CI systems, explore opportunities for explainability through token-level saliency, and describe remediation workflows, positioning this work as a scalable foundation for multi-language, repository-level analysis.

Keywords: Code Smell Identification, CodeBERT, Machine Learning, Deep Learning, Software Quality, Software Engineering

ACKNOWLEDGMENTS

Throughout this project, my advisor, Dr. Robert Chun, has been a huge source of guidance, inspiration, and support, and I am truly thankful to him. His comments and insights substantially changed the course and breadth of this study.

I would also like to sincerely thank Dr. Sayma Akther for her mentorship and consistent support during the course of this work. Her expertise and direction greatly enhanced the quality of the project. I am equally grateful to Nava Prashanth Kakarla for his collaboration and helpful technical discussions at various stages of the project.

I am also thankful to the Department of Computer Science at San José State University for providing the resources and academic environment necessary to conduct this work.

Lastly, I extend my appreciation to my peers, friends, and family for their moral support and patience during the course of this project.

LIST OF TABLES

List of Figures

List of Tables

1 Introduction

Rapid software system evolution puts constant pressure on development teams to produce features without sacrificing quality. Design shortcuts, redundant logic, large classes, and overworked methods eventually build up into ”code smells” [66]—symptoms that are associated with greater maintenance costs, lower readability, and higher defect rates [68]. Therefore, in contemporary software engineering, identifying and fixing code smells early on is essential to managing technical debt and maintaining developer productivity.

Conventional methods of smell detection depend on manual reviews or rule-based static analysis. These methods are helpful, but they have three enduring drawbacks. They are challenging to scale across very big and rapidly evolving codebases, to start. Second, manually created criteria could overflag harmless constructs or overlook semantically subtle patterns, creating noise and weariness for reviewers. Third, the subjective and uneven nature of manual processes affects reviewers, teams, and organizations [67, 68]. By using the semantic structure of programs rather than just surface metrics or heuristics, recent developments in representation learning for source code—particularly transformer-based models trained on huge code corpora—offer a chance to identify odors [65].

This capstone investigates a workable and repeatable workflow for Java code smell detection. To represent lexical and structural semantics, I used CodeBERT [65] to convert source code into 768-dimensional embeddings. I evaluated two complementary classifiers on top of these embeddings: a compact three-layer neural network as a flexible, learnt decision function and random forest as a robust, interpretable baseline. my research is based on the extensive SmellyCode++ dataset (107,554 samples) [64], and my experiments are designed to take into account practical limitations such as severe class imbalance, computational limitations, and the requirement for transparent evaluation and reproducibility.

1.1 Motivation

- Organizations need scalable, low-friction mechanisms to monitor code quality continuously.
- Smell detection can focus code reviews on high-risk regions and guide refactoring effort.
- Embedding-based models promise improved generalization beyond hand-tuned rules.
- A clear, reproducible baseline helps bridge research prototypes and production practice.

1.2 Problem Statement

How can I detect code smells in Java source files at scale using learned representations, offering transparent evaluation and practical suggestions for adoption? The work investigates binary smell identification (any smell vs. clean) with actual class imbalance, evaluating the accuracy of transformer-based embeddings compared to standard features and common classifiers [65].

1.3 Objectives and Research Questions

Objectives

1. Build an end-to-end pipeline that ingests code, extracts CodeBERT embeddings, trains classifiers, and produces auditable evaluations.
2. Compare Random Forest and Neural Network classifiers on identical embeddings.
3. Quantify performance with accuracy, precision, recall, F1-score, and confusion matrices; analyze failure modes.
4. Validate utility on representative Java snippets and discuss deployment pathways.

Research Questions

- RQ1: Do CodeBERT embeddings enable effective binary smell detection on SmellyCode++?
- RQ2: Given identical embeddeding features how do Random Forest and Neural Network compare?
- RQ4: What practices improve reproducibility and reduce threats to validity?

1.4 Report Organization

Section 2 reviews background literature on code smells, software metrics, and learning-based code analysis. Section 3 details the dataset, labels, and preprocessing. Section 4 presents exploratory analysis of distributions and correlations. Section 5 describes the methodology and evaluation design. Section 6 explains implementation details and reproducibility. Section 7 reports experiments and ablations. Section 8 discusses results, limitations, and practical implications. Section 9 concludes and outlines future work including multi-label detection and model fine-tuning.

2 Literature Review

This section gives a full view of what I study now about how to find code smells, how they work, what I do with them, and and how I use machine learning, deep learning, and transformers. I looked at how detection has grown and found things I need more study on that make this work hard.

2.1 Code Smells: Definition and Taxonomy

Code smells, first raised by Fowler [66], are clues on the surface of code that point to deeper issues in design that can cause maintenance problems, poor code quality, and tech debt. Unlike bugs, code smells do not stop code from working, but they suggest that code is hard to understand, change, or build on [66, 68].

Mäntylä et al. [4] did one of the earliest experiments on code smells, where they found different kinds and created a taxonomy that many others have used. They showed code smells are everywhere in real programs and that they make programs harder to keep going [4].

Yamashita and Moonen [68] studied how code smells and maintainability are related, finding that some smells like God Class and Long Method are more connected to how hard a program is to fix than others.

In this work, I looked at four smells that have been studied a lot and are usually seen as very important.

- **Long Method**: Methods that exceed a certain length threshold and perform multiple responsibilities, violating the Single Responsibility Principle [66]. Such methods are difficult to understand, test, and maintain [68].
- **God Class**: Classes that know too much or do too much, often characterized by excessive coupling, high complexity, and multiple responsibilities [66, 4]. These classes become maintenance bottlenecks.
- **Feature Envy**: Methods that are more interested in data from other classes than their own, indicating misplaced functionality [66, 68].
- **Data Class**: Classes that primarily contain data fields with minimal behavior, often serving as simple data containers [66, 4].

2.2 Traditional Detection Methods

At first, people used tools that checked code by rules or numbers they already set which can be called static analysis methods. These static analysis techniques can be grouped into two main types: metric-based and rule-based.

2.2.1 Metric-Based Approaches

Metric-based methods use software metrics such as lines of code (LOC), cyclomatic complexity, coupling, and cohesion to identify potential smells [7]. Marinescu [8] proposed metric-based detection strategies using threshold rules to detect design flaws. While easy to implement and interpret, these techniques often suffer from high false positive rates and limited adaptability across projects [9].

Complexity and OO metrics including Halstead metrics [10], McCabe's cyclomatic complexity [11], CBO and LCOM [12] have been widely adopted, yet they alone cannot capture the semantic and contextual aspects of code that are crucial for accurate smell detection [13].

2.2.2 Rule-Based Approaches

Rule-based techniques employ explicit heuristics to match known smell patterns. Tools such as PMD, Checkstyle, and SonarQube operationalize these rules [14]. Despite industrial adoption, they often generate many false positives, miss subtle or context-dependent smells, and struggle with subjectivity in smell definitions [68, 15, 16, 17]. Their rigidity also limits adaptability to evolving codebases and different languages.

2.3 Machine Learning Approaches

Limitations of traditional methods have motivated ML-based techniques that learn patterns from data and adapt to different contexts, improving accuracy and reducing false positives [18, 19].

2.3.1 Supervised Learning Methods

Supervised approaches train classifiers on labeled smell datasets. Fontana et al. [20] compared Decision Trees, Naive Bayes, and SVMs, showing ML can outperform metric-only baselines. Arcelli Fontana et al. [21] and Maiga et al. [22] further demonstrated strong performance for ensemble models, particularly Random Forest, on God Class and Long Method detection.

Recent work explores richer setups: Khleel and Nehéz [23] evaluated multiple algorithms with data balancing and reported strong performance (up to 100% accuracy for some smells). Bhuiyan et al. [24] combined advanced classifiers (e.g., XGBoost, AdaBoost) with

SMOTE, correlation-based feature selection, and hyperparameter optimization to achieve near-perfect results on selected datasets.

2.3.2 Ensemble Learning Methods

Ensemble methods (Bagging, Boosting, Voting) improve robustness by aggregating multiple models. Alazba et al. [25] evaluated ensembles for Python smell detection and reported high accuracy for Large Class and Long Method. Their success is attributed to better handling of imbalance and variance [26, 27], though at the cost of higher complexity and reduced interpretability.

2.3.3 Unsupervised Learning Approaches

To mitigate reliance on labeled data, unsupervised methods such as the approach by Gupta et al. [28] use feature engineering and clustering to infer smells. These can discover novel patterns [29, 30] but often require expert interpretation and may be less reliable than supervised models.

2.4 Deep Learning Approaches

Deep learning enables learning complex, non-linear patterns directly from code representations.

2.4.1 Neural Network Architectures

Agrawal et al. [31] showed that deep models can effectively capture semantic patterns for smell detection. Multi-layer perceptrons (MLPs) have been applied with promising results [23, 32], though they may struggle to fully exploit structural properties of code [33].

2.4.2 Recurrent Neural Networks

RNNs and LSTMs [34] have been investigated for modeling sequential aspects of source code [35]. While they capture order and context, they suffer from scalability issues and difficulty modeling long-range dependencies [36].

2.5 Transformer Models for Code Analysis

Transformer architectures [56] have transformed both NLP and code analysis.

2.5.1 CodeBERT and Pre-trained Models

CodeBERT [65] is a pre-trained transformer for programming and natural languages, trained on large code–text corpora. It has been successfully applied to tasks such as code search, documentation, and summarization [65, 37]. Its dual-modal pre-training captures both syntactic and semantic information, making it a strong candidate for smell detection where semantic cues are essential [38].

2.5.2 GraphCodeBERT

GraphCodeBERT [39] extends CodeBERT with data-flow graphs, enabling richer structural and semantic representations and achieving superior performance on several code understanding tasks.

2.5.3 Other Transformer-Based Models

Other models such as CuBERT [40], PLBART [41], and CodeGen [42] further demonstrate the potential of transformers for software engineering tasks, including bug detection, code completion, and repair.

2.6 Datasets for Code Smell Detection

2.6.1 SmellyCode++ Dataset

SmellyCode++ [64] is one of the largest datasets for smell detection, containing 107,554 Java samples annotated for Long Method, God Class, Feature Envy, and Data Class, along with 14 numerical metrics. Its strong class imbalance (about 9.22% smelly vs. 90.78% clean) reflects realistic distributions and highlights the need for robust imbalance handling methods [43].

2.6.2 Other Datasets

Other resources include DACOS [44] and multiple domain- or smell-specific datasets [25, 45, 46]. However, the lack of standardized, widely adopted benchmarks remains a barrier to fair comparison of methods [46].

2.7 Challenges and Limitations

Key challenges include:

2.7.1 Class Imbalance

Severe imbalance biases models towards the majority (clean) class [64, 43]. Techniques such as SMOTE, sampling, and class weighting [23, 24, 47, 48] mitigate but do not fully eliminate this issue.

2.7.2 Subjectivity and Context Dependency

Smell definitions are subjective and context-dependent [68, 16, 49], causing inconsistent labeling and hindering generalization [50].

2.7.3 Lack of Standardized Evaluation

Heterogeneous metrics, datasets, and protocols complicate comparisons across studies [46, 51, 52, 53].

2.7.4 Scalability and Computational Cost

Deep and transformer-based models require substantial computational resources [54, 55, 56], which can limit their practical deployment in continuous integration environments [56].

2.8 Systematic Literature Reviews

Systematic reviews by Azeem et al. [57] and Yadav et al. [58] synthesize ML-based smell detection research, highlighting recurring issues such as imbalance, subjectivity, and dataset inconsistency, and stressing the need for more comparable, reproducible studies.

2.9 Research Gaps and my Contribution

Despite extensive work, several gaps persist:

- Limited use of transformer-based models like CodeBERT specifically for smell detection [65, 38].
- Few thorough comparisons between traditional ML and deep learning approaches on common representations [59].
- Focus on binary detection rather than fine-grained or multi-label smell classification [60].

Our work addresses these gaps by:

1. Leveraging CodeBERT embeddings for smell detection and systematically assessing their effectiveness [65].

2. Comparing Random Forest and Neural Network models on identical embeddings to understand trade-offs [61].

3. Conducting large-scale, reproducible experiments on SmellyCode++ [64], providing an open, extensible framework.

3 Dataset and Methodology

In this section I will talk about the data set used in this work, how I preprocessed it, how I extracted features from it, and what machine learning models I used for code smell detection.

3.1 Dataset Description

The foundation of any machine learning study is the dataset used to analyze, interpret, and train the algorithms. this section examines the smellycode++ dataset, which is the primary data source for my code smell detection research. understanding the datasets properties, structure, and inherent qualities is essential for developing detection models and analyzing their performance.

3.1.1 Dataset Overview

The SmellyCode++ dataset is a large set of Java code samples labeled for code smells. This was chosen because of its size, variety, and good explanation that make it easy to study and test code smell detection ways. The dataset has many different Java programs, so there is enough variation to learn good machine learning models. This table 3.1 lists the main details of the dataset. With over 107,000 samples, this dataset has strong data power for study and testing.

Java code's focus fits my study because Java is still a top language used in both work and school. The smaller size of about 18.1 MB makes it easier to handle on computers but still has enough details to learn from.

Table 3.1: Dataset Overview

Property	**Value**
Dataset Name	SmellyCode++
Source	figshare (DOI: 10.6084/m9.figshare.28519385.v1)
Total Samples	107,554
Programming Language	Java
Total Features	22
Code Smell Types	4 (Long Method, God Class, Feature Envy, Data Class)
Memory Usage	$\sim$18.1 MB
Data Format	CSV

The data set has some good and bad points. There are 107554 samples in the set, which is a lot. The data set has 22 features in all. It also has four types of code smell: Long Method, God Class, Feature Envy, and Data Class. This list of smells covers a lot of ground in the world of coding.

3.1.2 Dataset Features

The dataset has many features that cover the structure and meaning of code. They can be grouped into three main categories: (1) code metrics based on software engineering, (2) information about the code samples, and (3) binary labels that show whether or not a code smell exists. Each feature plays a role in model development, and knowing how each feature is interpreted is crucial.

Table 3.2 provides a complete count of all 22 features of the dataset. Features 1-2 (File and Project) are identifiers and meta data, which can be used to trace back to original source code. Features 3-15 are different types of software metrics, including Halstead metrics (features 4-14) and other measures of complexity. The Halstead metrics were created by Maurice Halstead in the 1970s and they are in fact metrics on program complexity, based on operators and operands. They are another way to estimate how maintainable or understandable source code is.

Table 3.2: Complete Feature List

#	Feature Name	Type	Description
1	File	Object	File path of the source code
2	Project	Object	Project name
3	Logical Lines	Integer	Number of logical lines of code
4	Distinct Operators	Integer	Number of unique operators
5	Distinct Operands	Integer	Number of unique operands
6	Total Operators	Integer	Total count of operators
7	Total Operands	Integer	Total count of operands
8	Vocabulary	Integer	Sum of distinct operators and operands
9	Length	Integer	Total operators + total operands
10	Calculated Length	Float	Calculated program length (N)
11	Volume	Float	Program volume metric (V)
12	Difficulty	Float	Program difficulty metric (D)
13	Effort	Float	Program effort metric (E)
14	Time Required	Float	Estimated time to understand code
15	Bugs	Float	Estimated number of bugs
16	Cyclomatic Complexity	Integer	McCabe's cyclomatic complexity
17	Class	Object	Class name
18	Code	Object	Java source code text
19	Long method	Integer	Label (0 or 1)
20	God class	Integer	Label (0 or 1)
21	Feature envy	Integer	Label (0 or 1)
22	Data class	Integer	Label (0 or 1)

The list of features includes many important ones for finding code smell. Cyclomatic Complexity (feature 16) was made by Thomas McCabe and shows how many paths are inside a program. This number links to how hard the code is to understand. The Logical Lines number (feature 3) is simple and shows how big the code is. The Halstead numbers also give good ways to see code that might be hard to read. Feature 18 (Code) shows the text of the Java code itself so that transformer based models such as CodeBERT can understand the meaning.

3.1.3 Code Smell Distribution

Knowing how code smells are spread out in the dataset is key for many reasons. First, it tells us how common each smell type is. This helps us see how much it really matters to find each smell type. Second, how smells are spread out can show us if there are issues with how I divided the data into groups (class imbalance). Third, looking at how smells go together can help us see how smells are related to each other.

Table 3.3 shows how many times I saw each smell in the whole dataset. I saw a lot more

clean code (90.78%) than code with smells (9.22%). This is often seen in real-world software projects. Most code is pretty clean. The code with smells is rare and needs fixing. Of the four smell types, God Class was the most common with 4.03%. Data Class was next with 3.05%. Feature Envy had 1.86% and Long Method had 1.46%.

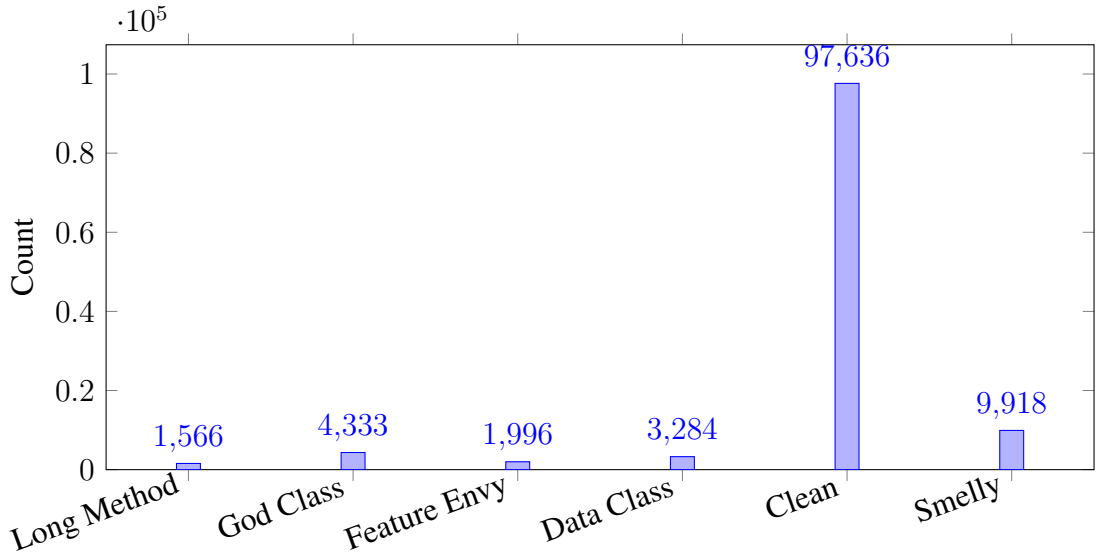
Table 3.3: Code Smell Distribution in Full Dataset

Code Smell Type	**Count**	**Percentage**	**Description**
Long Method	1,566	1.46%	Methods that are too long and perform multiple responsibilities
God Class	4,333	4.03%	Classes that know too much or do too much
Feature Envy	1,996	1.86%	Methods more interested in other classes than their own
Data Class	3,284	3.05%	Classes that primarily contain data with minimal behavior
Total Smelly	**9,918**	**9.22%**	Samples with at least one code smell
Clean	**97,636**	**90.78%**	Samples with no code smells
Total	**107,554**	**100.00%**	Complete dataset

The distribution patterns reveal important insights. God Class being the most common smell suggests that classes with excessive responsibilities are a frequent issue in Java codebases, possibly due to evolutionary design where classes accumulate functionality over time. The relatively low prevalence of Long Method (1.46%) might indicate that developers are generally aware of method length guidelines, or that automated refactoring tools help maintain method sizes. The total of 9,918 smelly samples represents a substantial number for training purposes, though the imbalance requires careful handling through sampling strategies.

Figure 3.1 provides a visual representation of the code smell distribution, making the imbalance immediately apparent. The bar chart clearly shows the dominance of clean code samples, with the smelly samples appearing as a small fraction. This visualization helps contextualize the challenge of code smell detection: models must learn to identify relatively rare patterns within a predominantly clean codebase.

Figure 3.1: Code Smell Distribution Visualization



The chart shows absolute counts. Percentages: Long Method 1.46%, God Class 4.03%, Feature Envy 1.86%, Data Class 3.05%, Clean 90.78%, Smelly 9.22%.

3.1.4 Software Metrics Overview

The software metrics in the dataset offer quantitative measures of code traits that are the building blocks of code smell detection. The metrics measure different aspects of code complexity, size, and maintainability, each giving a different perspective on code quality. Knowledge of the statistical properties of these metrics is important for feature engineering, normalization, and model interpretation.

Table 3.4 shows detailed statistical summaries of five main metrics: Logical Lines, Cyclomatic Complexity, Volume, Difficulty, and Effort. The stats show many significant patterns. The mean of 45.2 for Logical Lines with a stdev of 38.7 shows a lot of variation in code size with some methods being very long. The median of 35 lines shows most code samples are fairly short and small and the max of 302 lines shows very long code samples that are outliers and most likely code smells.

Table 3.4: Statistical Summary of Key Metrics

Metric	**Mean**	**Std Dev**	**Min**	**P25**	**Median**	**P75**	**Max**
Logical Lines	45.2	38.7	1	18	35	58	302
Cyclomatic Complexity	8.3	7.2	1	3	6	11	77
Volume	245.8	198.4	12.5	98.3	198.7	342.1	1,245.6
Difficulty	12.4	9.8	1.2	5.6	9.8	16.4	58.3
Effort	3,045.2	3,124.5	15.0	550.5	1,947.2	5,612.8	72,456.1

The Cyclomatic Complexity stats show a mean of 8.3 with a median of 6, which indicates that most code samples are of moderate complexity. Yet the maximum of 77 code is extremely complex and would be hard to change or test. The Halstead metrics (Volume, Difficulty, Effort) showed similar kinds of distributions that were skewed towards the lower end, but with some high outliers. The Effort metric was especially skewed with a minimum of 15.0 and a max of over 72,000. These large spread measures reflected the large range of code in the data set.

These stats have important implications for model building. The skewness suggests that normalization or log transformation may help. Outliers mean robust processing steps will be needed to keep the models from being overly affected by any extreme values. The variability in the measures shows that the code samples varied in their characteristics, but in a way that would help us train a good model.

3.2 Data Preprocessing

The data preprocessing is a key step in the process of machine learning pipeline and it converts raw data into usable data to train a model. For code smell detection, the preprocessing had three major steps, which are turning the multi-label target into binary problem, addressing class imbalance by sampling, and data preparation for the feature extraction. This section describes each step of the preprocessing, and the justification of the approach used.

3.2.1 Target Variable Creation

The original dataset has four separate binary labels, one for each smell type. Although multi-label classification is possible, I converted this to a binary classification problem in which the target variable indicates whether the code sample contains any smell or is clean. This conversion makes the problem easier while keeping its practical relevance. In practice, developers are often interested in any code smell, regardless of its type. This change has several advantages.

First, it increases the number of positive samples as it combines all smell types. Second, it makes the application more straightforward for real-life cases where the developer wants a general indicator for code quality. Third, it simplifies the evaluation and explanation of the model. Nonetheless, this change loses information on specific smell types, which could be explored in future work via multi-label classification. Table 3.5 depicts the logic of the transformation.

Any sample with at least one code smell (regardless of its type or combination of types) is labeled as smelly (1), while only samples with all zeros are labeled as clean (0). This approach considers all code smells as equally important indicators of code quality problems.

Table 3.5: Target Variable Transformation

[H]

Original Labels	**Combined Label**	**Description**
All zeros (0,0,0,0)	0 (Clean)	No code smells detected
Any one (1,0,0,0), (0,1,0,0), etc.	1 (Smelly)	At least one code smell present
Multiple (1,1,0,0), (1,1,1,0), etc.	1 (Smelly)	Multiple code smells present

Table 3.6: Target Variable Distribution

Class	**Count**	**Percentage**
Clean (0)	97,636	90.78%
Smelly (1)	9,918	9.22%
Total	**107,554**	**100.00%**

3.2.2 Sampling Strategy

Class imbalance is one of the most common challenges in machine learning, particularly in domains like code smell detection where positive cases (code smells) are relatively rare. Several strategies exist to address this issue: oversampling the minority class, undersampling the majority class, using class weights, or employing specialized loss functions. For this study, I employed stratified random sampling to create a balanced subset of the full dataset.

There are several reasons why I chose a balanced sample instead of the full imbalanced dataset. First, training with the full dataset would be computationally expensive. In particular, training on transformer-based features would be very slow. Second, balanced datasets help models learn to tell the two classes apart, as opposed to always predicting the majority class. Third, a balanced sample of 5,000 samples provides enough data for training while still being small enough for fast training.

Table 3.7 shows my sampling approach. I used stratified random sampling (with a random seed of 42) to guarantee that my samples would be reproducible. The balanced sample contains 2,500 clean samples and 2,500 smelly samples, which creates a class distribution of 50-50. This class distribution helps models learn from equal representation of the two classes, which should help them identify code smells.

Table 3.7: Sampling Strategy

Stage	Description	Sample Size	Rationale
Full Dataset	Original SmellyCode++ dataset	107,554	Complete dataset
Balanced Sample	Stratified random sampling	5,000	Computational efficiency
- Clean samples	Random sample from clean class	2,500	Balanced representation
- Smelly samples	Random sample from smelly class	2,500	Balanced representation
Random Seed	For reproducibility	42	Ensures consistent results

Table 3.8: Balanced Sample Distribution

Class	Count	Percentage
Clean (0)	2,500	50.00%
Smelly (1)	2,500	50.00%
Total	**5,000**	**100.00%**

3.2.3 Train-Test Split

Proper data splitting is essential for reliable model evaluation. The train-test split separates data into training and testing sets, ensuring that model performance is evaluated on unseen data. This separation prevents overfitting and provides realistic estimates of how models will perform in production. For this study, I employed a stratified 80-20 split, which maintains the balanced class distribution in both training and testing sets.

The 80-20 split is a common choice in machine learning, providing sufficient data for training (80%) while reserving a substantial portion for evaluation (20%). Stratification ensures that both splits maintain the same class distribution as the balanced sample, preventing one split from having a different class balance that could skew results.

Table 3.9 shows the resulting split sizes. The training set contains 4,000 samples (2,000 clean, 2,000 smelly), providing ample data for model learning. The test set contains 1,000 samples (500 clean, 500 smelly), which is sufficient for reliable performance estimation while maintaining statistical significance.

Table 3.9: Train-Test Split Configuration

Split	Size	Clean Samples	Smelly Samples	Percentage
Training	4,000	2,000	2,000	80%
Testing	1,000	500	500	20%
Total	**5,000**	**2,500**	**2,500**	**100%**

Table 3.10: Split Parameters

Parameter	Value	Description
Test size	0.2 (20%)	Proportion of data for testing
Random state	42	Ensures reproducibility
Stratify	Yes	Maintains class distribution
Shuffle	Yes	Randomizes sample order

3.3 Feature Extraction

Feature extraction refers to converting raw code into numeric representations that machine learning models can take in. Traditional methods rely on manually designed features like code metrics but recent progress in natural language processing has made it possible to extract semantic features with transformer models. This section discusses my method for extracting semantic embeddings from source code with CodeBERT, a pre-trained transformer model built specifically for code understanding.

3.3.1 CodeBERT Embedding Extraction

CodeBERT is a model that uses a special type of deep learning called a bidirectional encoder transformer. It was built with the RoBERTa design and trained on lots of code and text. Unlike old ways of measuring code, CodeBERT places the code in context. So it understands not only what the code means but how it relates to the surrounding code. With this, the model can find code smells based on what the code means and its structure, not only on numbers.

The process of getting the embedding turns raw Java code into a fixed size of numbers, which contains meaning. These embeddings act as the input for other models to classify. I used CodeBERT instead of other ways to embed code, because it has been shown to do a good job at understanding code and can find patterns both in how code is written and what it means.

Table 3.11 shows the setting used for getting the embedded ones. The architecture of the model has 12 transformer layers with 768 dimensional embedings for each code sample. The maximum length of sequence is set at 512 tokens because it gives a good trade off between getting enough context from the code sample and run-time complexity. Longer code samples are truncated, whereas short ones are padded so the input sizes are all the same.

Table 3.11: CodeBERT Configuration

Parameter	Value	Description
Model	microsoft/codebert-base	Pre-trained transformer model
Architecture	RoBERTa-based	12-layer transformer encoder
Embedding Dimension	768	Dimensionality of output vectors
Max Sequence Length	512 tokens	Maximum input length
Tokenization	Byte-Pair Encoding (BPE)	Subword tokenization
Pooling Strategy	CLS token	Uses [CLS] token representation

Table 3.12: Embedding Extraction Process

Step	Description	Output
1	**Code Preprocessing**	Clean and normalize Java code
2	**Tokenization**	Convert code to token IDs (max 512 tokens)
3	**Padding/Truncation**	Ensure uniform length
4	**Model Forward Pass**	Extract hidden states from CodeBERT
5	**CLS Token Extraction**	Extract [CLS] token embedding (768-dim)
6	**Batch Processing**	Process multiple samples efficiently
7	**Storage**	Save embeddings as NumPy array

Table 3.13: Embedding Extraction Statistics

Metric	Value
Total samples processed	5,000
Embedding dimension	768
Total embedding size	(5,000, 768)
Processing time	$\sim$19.2 minutes
Batch size	8 samples
Total batches	625
Average time per batch	$\sim$1.84 seconds

3.3.2 Rationale for CodeBERT

The decision to use CodeBERT for feature extraction was based on a number of strong benefits it had over traditional feature engineering approaches. While code metrics give a good quantitative measure, they are often not enough to get the semantic nuances and contextual relationships needed to find code smells. CodeBERT closes this gap by using pre-trained knowledge learned from large code corpora and understanding code semantics.

Table 3.14 lists the main benefits of using CodeBERT embeddings. The semantic understanding benefit is especially relevant for code smell detection because smells are often characterized by the code intent and design patterns and not by a simple numerical threshold. For example, Long Method is not only about the number of lines but whether the method has more than one purpose or responsibility which is something that can be only inferred with the semantic purpose of the method.

Table 3.14: Advantages of CodeBERT Embeddings

Advantage	**Description**
Semantic Understanding	Captures meaning and intent of code, not just syntax
Pre-trained Knowledge	Leverages knowledge from large code corpus
Context Awareness	Understands relationships between code elements
Language-Agnostic Features	Can be adapted to other languages
State-of-the-Art Performance	Proven effective in code understanding tasks
No Manual Feature Engineering	Eliminates need for domain-specific features

The pre trained info in CodeBERT is from training on millions of lines of code and it knows patterns, common words and bad practices. This transfer of knowledge is useful when you have less data to train on because the program already knows how code works. When context is needed, CodeBERT can tell how parts of code relate to each other, like how methods change a class or how variables are used.

Furthermore, CodeBERT does not require hand crafting features which is slow and it can miss key details. It lets the model learn features on its own from the code text and it might see patterns a human engineer does not. The language-agnostic design means it could be used for other languages with little change.

3.4 Model Architectures

Your choice of model shape can affect how quickly it operates, how well it finds objects, and how simple it is to grasp. Two related approaches are used in this work. The first is Random Forest, a collection of decision trees that functions well and is easy to understand. The second kind of AI is a neural network, which is capable of identifying complex patterns. This section describes both kinds of models, their training and setup procedures, and the benefits of each.

3.4.1 Random Forest Classifier

Random Forest is an ensemble learning method that constructs multiple decision trees during training and outputs the mode of the classes (for classification) predicted by individual trees. This ensemble approach reduces overfitting and improves generalization compared to single

decision trees. Random Forest is particularly well-suited for code smell detection because it can handle high-dimensional feature spaces, provides feature importance scores for interpretability, and is relatively robust to outliers and noise in the data.

The Random Forest algorithm works by training each tree on a bootstrap sample of the training data and considering only a random subset of features at each split. This randomness decorrelates the trees, making the ensemble more robust. The final prediction is made by aggregating predictions from all trees, typically through majority voting for classification tasks.

Table 3.15 specifies the hyperparameters used for the Random Forest classifier. The setting is 100 trees, which gives a good mix of how hard it is and how long it takes. Each tree is not allowed to be bigger than 20 levels deep, so it does not get too big and overfit, but it can still find complex stuff in the 768 size embed space.

Table 3.15: Random Forest Hyperparameters

Hyperparameter	**Value**	**Description**
n_estimators	100	Number of decision trees
max_depth	20	Maximum depth of each tree
random_state	42	Seed for reproducibility
n_jobs	-1	Use all available CPU cores
criterion	gini	Splitting criterion (default)
min_samples_split	2	Minimum samples to split node (default)
min_samples_leaf	1	Minimum samples in leaf node (default)
max_features	sqrt	Features considered for split (default)
bootstrap	True	Bootstrap sampling (default)

Table 3.16: Random Forest Advantages and Limitations

Aspect	Description
Advantages	• Interpretable feature importance • Handles imbalanced data well • Fast training and inference • Robust to overfitting • No feature scaling required • Handles non-linear relationships
Limitations	• May struggle with very complex patterns • Limited ability to capture sequential dependencies • Less effective for high-dimensional sparse data • Memory intensive for large datasets

However, Random Forest has limitations in capturing very complex patterns that might require deep hierarchical feature learning. The model may also struggle with sequential dependencies in code, though this is less relevant when using pre-computed embeddings. Despite these limitations, Random Forest serves as an excellent baseline and provides valuable interpretability that complements the neural network approach.

3.4.2 Neural Network Classifier

Neural networks offer a complementary approach to Random Forest, capable of learning complex non-linear relationships through multiple layers of transformations. Unlike Random Forest, which makes explicit feature splits, neural networks learn hierarchical feature representations automatically. This makes them particularly well-suited for high-dimensional embedding spaces where complex interactions between features may be important for classification.

The neural network architecture used in this study is a feedforward multi-layer perceptron (MLP) that processes the 768-dimensional CodeBERT embeddings. The architecture is designed to progressively reduce dimensionality while learning increasingly abstract representations of code semantics. Dropout regularization is employed to prevent overfitting and improve generalization.

Table 3.17 details the network architecture. The input layer receives 768-dimensional CodeBERT embeddings, which are then processed through two hidden layers with 256 and 128

units respectively. This reduction in size helps the net learn more of the harder ideas. ReLU (Rectified Linear Unit) makes the net learn without being flat.

Table 3.17: Neural Network Architecture

Layer	**Type**	**Units**	**Activation**	**Dropout**	**Parameters**
Input	Dense	768	-	-	CodeBERT embeddings
Hidden 1	Dense	256	ReLU	0.3	196,864
Hidden 2	Dense	128	ReLU	0.3	32,896
Output	Dense	2	Softmax	-	258
Total Parameters					**230,018**

Table 3.18: Neural Network Training Configuration

Hyperparameter	**Value**	**Description**
Optimizer	Adam	Adaptive learning rate optimizer
Learning Rate	0.001	Initial learning rate
Weight Decay	1e-5	L2 regularization coefficient
Loss Function	Cross-Entropy	Binary classification loss
Epochs	20	Number of training iterations
Batch Size	32	Samples per training batch
Device	CPU	Computing device (CUDA available)
Random Seed	42	For reproducibility

Table 3.19: Neural Network Training Progress

Epoch	**Training Loss**	**Training Accuracy**	**Description**
5	0.4959	74.78%	Early training phase
10	0.4621	76.50%	Mid training phase
15	0.4585	76.40%	Convergence phase
20	0.4417	77.97%	Final model

Table 3.20: Neural Network Advantages and Limitations

Aspect	Description
Advantages	• Can learn complex non-linear patterns • Adapts to data characteristics • Good generalization with dropout • Scalable to larger architectures • Can be fine-tuned for specific tasks
Limitations	• Less interpretable than Random Forest • Requires more computational resources • Longer training time • Sensitive to hyperparameters • Requires careful initialization

The neural network's dependence on hyperparameters makes fine adjustment necessary, and the longer time to train compared to the Random Forest is a disadvantage, but the neural network can grow larger which makes it good for future use in this work.

3.5 Experimental Design

A good test setup is important to see what works and what doesn t work between models. Here I show how I evaluated the models by explaining what measurements I took to see how well they perform, what steps I took to test them, and why I designed it that way. My test setup makes sure that the models are compared fairly to each other, and that you can learn a lot about what each model does well, and what it does not.

3.5.1 Evaluation Metrics

Picking the right evaluation scores is key to finding the best way to test the system, especially for binary tests that may have uneven classes. Each score tells us different things: accuracy shows the total correctness, precision shows how much of the positive results are right, recall shows how many of the positives I got, and the F1-score balances precision and recall. For code smell tests, all these scores matter, as some cases might need one score over another.

Tabl 3.21e shows the metrics used in this work. Accuracy gives an overall score of right and wrong guesses but can be bad in imbalanced data if you are not careful. Precision is important when finding code smell because if it says a clean piece of code is smelling that is a waste of a developer's time and makes the tool less trusted. Recall is also important because if it does not find a real code smell then it is not doing its job.

Table 3.21: Evaluation Metrics

Metric	**Formula**	**Description**
Accuracy	(TP + TN) / (TP + TN + FP + FN)	Overall correctness
Precision	TP / (TP + FP)	Proportion of positive predictions that are correct
Recall	TP / (TP + FN)	Proportion of actual positives correctly identified
F1-Score	$2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$	Harmonic mean of precision and recall
Confusion Matrix	–	Detailed breakdown of predictions

Where TP = True Positives, TN = True Negatives, FP = False Positives, FN = False Negatives.

The F1-score is a good tool because it mixes precision and recall, so it helps to see how models do both jobs at once. The confusion matrix shows more details about true positives, true negatives, false positives, and false negatives, which helps to look at where models get it right and wrong. Having all this detail makes it easier to see if models have a bias, like if they tend to miss some code types more than others.

3.5.2 Experimental Procedure

The way I do my test is like a line of steps from start to end. Every step makes the next one easy. Data goes right from one part to the next: from cleaning and changing it, to getting the features, to teaching the machine, and finally to checking how good the machine is. This easy list of steps makes sure you can do the test again and find any problem at any part of the list.

Table 3.22 shows the full process of the experiment. The steps start with loading the full SmellyCode++ dataset, which is used for everything else. Preprocessing turns the multi-label problem into binary classification and balances the sample to fix the class imbalance problem of the original dataset. Extracting CodeBERT embeddings turns raw code into vectors of meaning, making it easy to use pre-trained knowledge about code.

Table 3.22: Experimental Steps

Step	Description	Output
1	**Dataset Loading**	Load SmellyCode++ dataset (107,554 samples)
2	**Data Preprocessing**	Create binary target variable, balanced sampling
3	**CodeBERT Embedding**	Extract 768-dimensional embeddings (5,000 samples)
4	**Train-Test Split**	80/20 stratified split
5	**Random Forest Training**	Train on 4,000 samples
6	**Neural Network Training**	Train on 4,000 samples (20 epochs)
7	**Model Evaluation**	Evaluate both models on 1,000 test samples
8	**Performance Comparison**	Compare metrics and analyze results
9	**Real-World Testing**	Test on sample Java files

The train test split makes sure the models see new data for a better idea of how they will perform. Both the Random Forest and Neural Network models are trained on the same data, so they can be compared fairly. The models are checked on the test data and give good results. Testing on actual Java code files shows the models work well in real life. This full process makes sure the models are not only good on the test data but also useful in real situations.

3.6 Implementation Details

The hardware and software that run this project are very important for others to get the same results and to do better or worse in performance. This part list the programs, hardware and choices of implementation that help others to run the experiments. For others to get the same results, this documentation of implementation environment is critical, and it can also show what parts of the experiments results can vary.

3.6.1 Software Environment

In this work, the programming environment is based on Python 3.8+ and includes several libraries for machine learning, deep learning, and data handling. The Python programming environment supports the implementation of neural networks and loading of the CodeBERT model. The Transformers library includes pre-trained CodeBERT models and tools for tokenization and getting embeddings.

Table 3.23 shows all the software parts and what they do. scikit-learn has tools for machine learning like Random Forest, making it easy to train data, split data for testing, and see how well the model works. Pandas and Numpy work on data and numbers. Matplotlib and

Seaborn help to see the data and share results. Using the newest versions of these tools will give you the newest fixes and updates, but setting specific versions would make it even more exact.

Table 3.23: Software and Library Versions

Component	Version	Purpose
Python	3.8+	Programming language
PyTorch	2.9.0	Deep learning framework
Transformers	Latest	CodeBERT model
scikit-learn	Latest	Machine learning utilities
Pandas	Latest	Data manipulation
NumPy	Latest	Numerical computations
Matplotlib	Latest	Visualization
Seaborn	Latest	Statistical visualization

3.6.2 Hardware Configuration

The hardware specs impact the computational time and can change the results due to variations in floating point precision or generation of random numbers. Modern CPUs and GPUs usually give same results, but recording the hardware helps other researchers find what might cause differences and lets them calculate how much resources it may take to repeat the work. Table 3.24 shows the hardware setup. All work was done on CPU, even though CUDA was possible. This decision makes it easier for more people to use and copy, since not all labs have GPUs.

But using a GPU would cut down the training time a lot, especially for neural net training and CodeBERT embedding pulls. The 4GBplus RAM rule makes sure there is enough memory to load the data, embeddings, and models all at once.

Table 3.24: Hardware Specifications

Component	Specification	Notes
CPU	Multi-core processor	Used for all computations
GPU	Not used	CUDA available but not used
RAM	4GB+ recommended	Sufficient for dataset and models
Storage	1GB+	For dataset, models, and embeddings

Storage requirements are relatively modest, with 1GB+ sufficient for the dataset, pretrained models, extracted embeddings, and trained model checkpoints. The use of CPU for all computations means that results should be reproducible across different hardware configurations, as long as the software versions are consistent.

3.7 Reproducibility

Reproducibility is a key part of science. It helps others check your work and make new things from it. In machine learning, reproducibility is hard to get because algorithms, data, and models all have randomness in them. This part lists all things that can be random and the seeds used to stop that. This lets you run experiments again and get the same results.

There are many places where randomness can come from in ML: sampling data, dividing data into train and test, making trees in a random forest, setting initial weights in neural networks, and many random operations in NumPy and PyTorch. Sometimes randomness helps the model, but if you don't control it, no one can get the same results again. Fixed random seeds fix that by making sure all the random things give the same values in any run.

Table 3.25 logs all random seeds used in the experiment pipeline. By using the same seed (like 42) across the entire process, others can replicate the study exactly. Someone who reads this and uses the same seed for sampling, splitting, or constructing trees will get the same results. The seed log runs through all the randomness at the data level (sampling, splitting) and the model level (tree building, weight initialization).

Table 3.25: Reproducibility Settings

Setting	**Value**	**Purpose**
Data Sampling Seed	42	Consistent sample selection
Train-Test Split Seed	42	Consistent data splits
Random Forest Seed	42	Consistent tree construction
Neural Network Seed	42	Consistent weight initialization
NumPy Random Seed	42	Consistent random operations
PyTorch Random Seed	42	Consistent tensor operations

Not just the seed but also the software version the data the way the experiment is done are factors for reproducibility. While I talk about how randomness can be controlled here in the same section I talk about software environment and experimental procedure and those provide the rest of reproducibility information needed for you to do the entire study over.

4 Exploratory Data Analysis

4.1 Introduction

This chapter is all about exploring the dataset I will use to detect code smells. Exploratory Data Analysis is an important first step that helps us understand the data before training models. With visual graphs and math, I see how code smells spread out, how the metrics for complexity form patterns, how different values relate to each other, and how clean code differs from code that smells. All this helps us make choices about preparing the data, which features to use, and which models to pick.

The aim of this EDA is three parts: (1) to see how different smell types are spread and how common they are, (2) to analyze how the numbers of code difficulty behave, and (3) to find links and signs that help us build the best model. I can get good clues about what to do next when I know my data well and can choose the right samples, the right things to show in my features, and the right model design that fits my data best.

4.2 Code Smell Distribution

Knowing how code smells spread is key to making good detection models. This dataset has more than one type of code smell and the types are god class data class feature envy long method each type shows some different design problems in code and knowing how they spread shows which ones are most common how seriously they are spread and if they can be related to other smells. This is important to know so I can make good sampling plans and so I understand the real importance of finding each smell type.

The distribution summary is very lopsidedwhich is normal for real software repositories where most code is quite clean. That makes it hard but also a good opportunity to see what makes code clean or smelly. The following images tell you more about what the distribution looks like.

4.2.1 Absolute Counts

Figure 4.1 shows the raw number of each type of code smell, giving a visual guide to their relative occurrence. The chart confirms that God Class is the most frequent code smell with 4,333 instances, almost three times more common than Long Method, which appears in only 1,566 samples. This large difference suggests that God Class might be a more common issue in Java codebases, possibly because classes tend to gather more tasks over time as the code grows.

The visualization also makes it very clear how common clean code samples are (97,636) versus any single smell type. This big gap makes it hard for models to pick up on the less common patterns. The total of 9,918 smelly samples (all smell types added together) is a large number for training but still only 9.22 of the data, so class imbalance must be handled carefully.

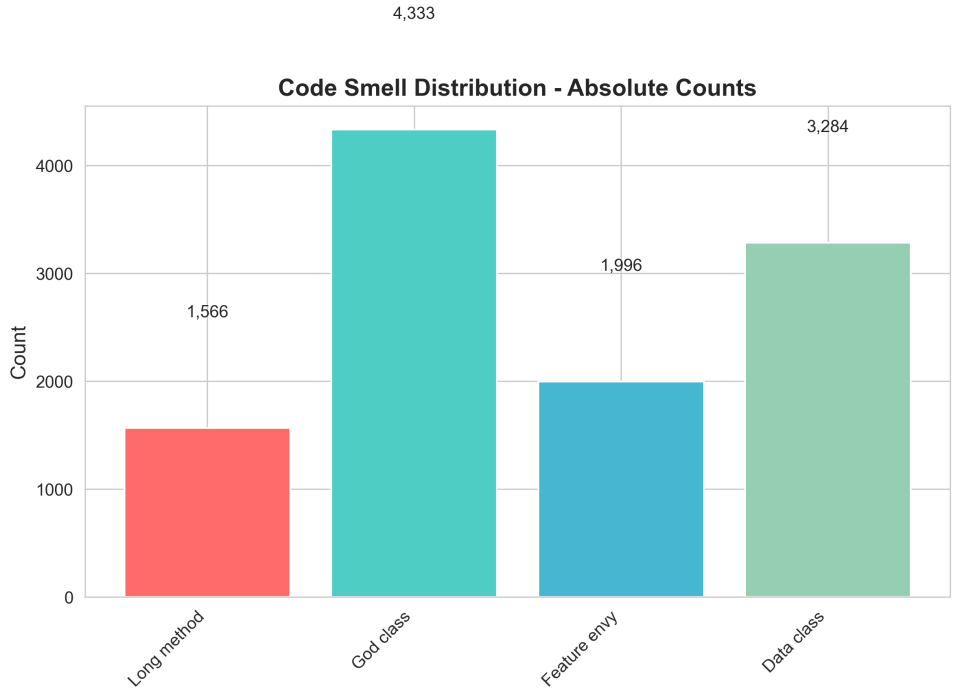


Figure 4.1: Code Smell Distribution - Absolute Counts. God Class is the most prevalent smell (4,333 samples), while Long Method is the least common (1,566 samples). The chart clearly illustrates the dominance of clean code samples and the relative rarity of code smells.

4.2.2 Percentage Distribution

While total counts are concrete figures, percentage counts give a better sense of how common each smell type is. Figure 4.2 shows that all individual code smells are less than 5 of the full set, with God Class being the highest at 4.03. This percentage view supports the class imbalance challenge and helps place each smell type in context of its practical significance. The percentage data is very helpful for understanding the scale of this detection problem. For example, if a model reports 95 accuracy, you might be wowed, but when 90.78 of the samples are clean, a trivial predictor that always says clean would get close to that performance.

This is why better evaluation metrics than accuracy, like precision and recall, matter.

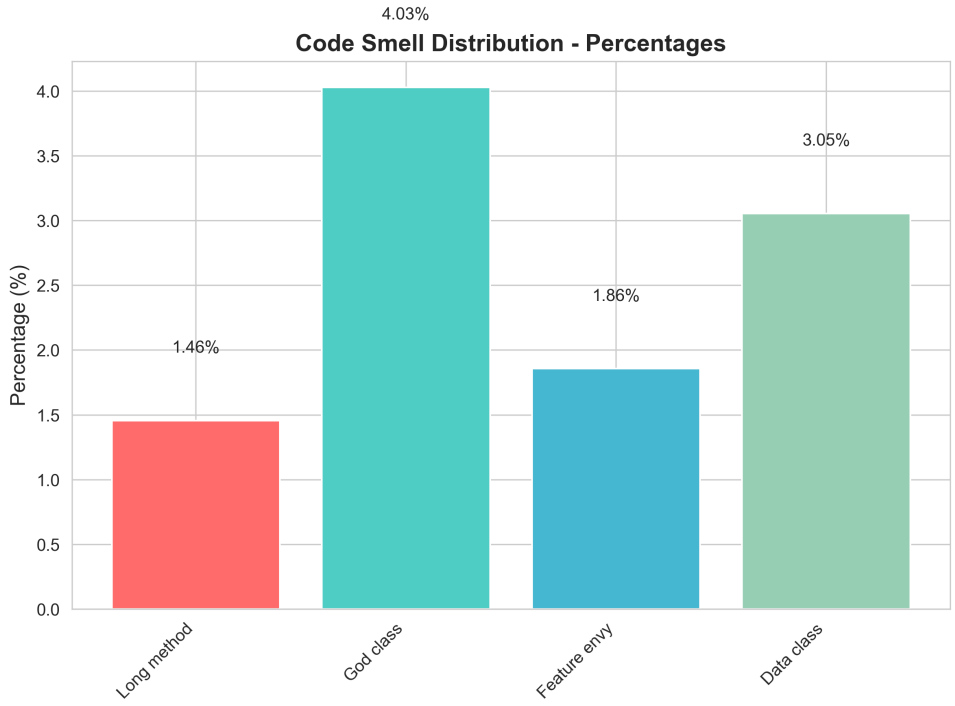


Figure 4.2: Code Smell Distribution - Percentages. All code smells represent less than 5%, highlighting significant class imbalance. This visualization emphasizes the rarity of code smells relative to clean code, which has important implications for model training and evaluation strategies.

4.2.3 Overall Class Distribution (Clean vs Smelly)

The yes or no question for code smells is easy the true problem is not which code smell it is but if there is a code smell at all. Figure 4.3 shows the binary yes or no distribution in a pie chart, making the severe class imbalance obvious The 90.78 clean code versus 9.22 smelly code is about a 10:1 ratio which is large but not huge a lot of imbalanced real world datasets go much higher.

This view helps you see why I needed to do balanced sampling for training the model. Without balance, the models would be mostly or all heavy leaning toward the main version. Using a pie chart makes it easy to see how much each side takes up, and here it clearly shows that the smelly code is a small minority that still counts a lot when you're making your model.

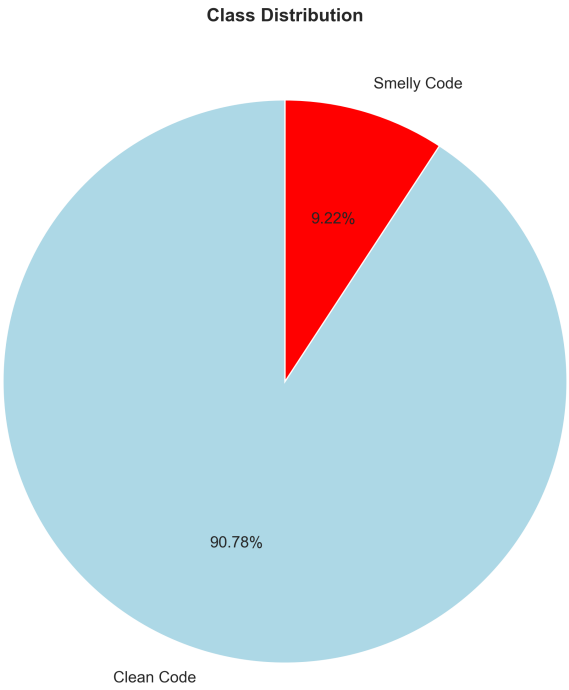


Figure 4.3: Class Distribution Pie Chart: Clean Code (90.78%) vs Smelly Code (9.22%). This visualization clearly illustrates the severe class imbalance, with clean code dominating the dataset. The approximately 10:1 ratio motivates the use of balanced sampling strategies for model training.

4.3 Complexity Metrics Distribution

Metrics that measure code complexity tell us about many parts of code that are related to smells of code. If I look at how different complex code tends to be I can see what is normal, what stands out as different, and what is linked to better or worse code. This part talks about five different types of measurements of code that are about how complex code is. These are Logical Lines of Code, Cyclomatic Complexity, and three Halstead measures of complexity: Volume, Difficulty, and Effort. These kinds of measurements show different parts of code that makes code complex and how many of these measurements different parts of code can have.

Checking what the distributions of complexity measures look like has many uses. First, it can show us which measures have unusual and large values that might need special care in processing. Second, it can tell us what typical ranges of complexity are in the data, helping us set useful thresholds for detection. Third, it can help us understand what features are most useful to a model's predictions and its feature importance scores. Last, it can also show us what parts of the distribution are most different in smelly and clean code, pointing out which metrics are most useful.

4.3.1 Logical Lines of Code

Logical Lines of Code LLOC is one of the easiest and best measures of code quality it tells us how many statements are in a sample of code it counts statements not physical lines of code or lines of code lines that you can see on the screen It is related to how big the code is and how easy it is to read or understand.

Figure 4.4 shows how many lines of code there are per sample. its skewed right meaning most samples dont have that many lines of code less than 60 but a few samples have way more lines of code more than 60. this is normal for software code where most methods are not super long but some methods are way too long which can be a problem and is a type of code smell called long method.

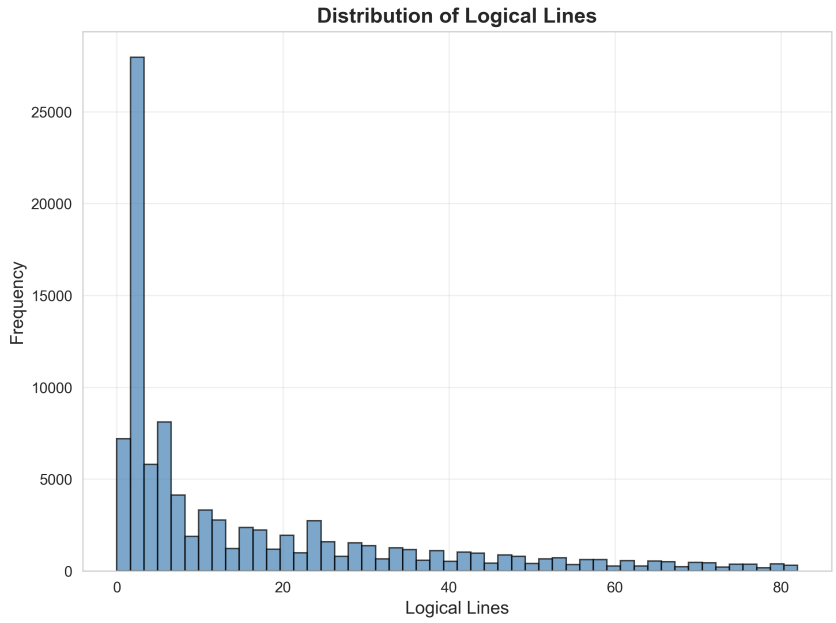


Figure 4.4: Distribution of Logical Lines of Code. The distribution is right-skewed, with most samples under 60 lines, indicating that the majority of code follows good practices regarding method length. The long tail represents outliers that may correspond to code smells, particularly Long Method instances.

This skewed shape has big effects. A lot of code pieces are normal in length. This means most programmers follow the rules for how long a method should be. But the ones in the tail of the graph are different. These are likely bad code that is too long. These weird ones are good for machine learning. The model can learn from code that is clearly too long.

4.3.2 Cyclomatic Complexity

Cyclomatic Complexity developed by Thomas McCabe tells how many ways you can go through the code of a program. It can be found out by how many control flow statements like ifs, fors, whiles, and switch cases are in there. The higher it is, the more choices you have

to get from one point in the code to the next. This makes the code much harder to understand, test, and fix. This is mostly used for code smell detection because if the code has a lot of control flow then there is usually a design problem.

Figure 4.5 shows most code samples are not too hard but have a level around 3 6. This range is okay since it means the code is easy to follow with just a few yesno points. But the data says there are some samples with values up to 77 which is too hard to keep up with and test. These high levels of difficulty are good to mark as code smells since they almost always break the rules for complexity.

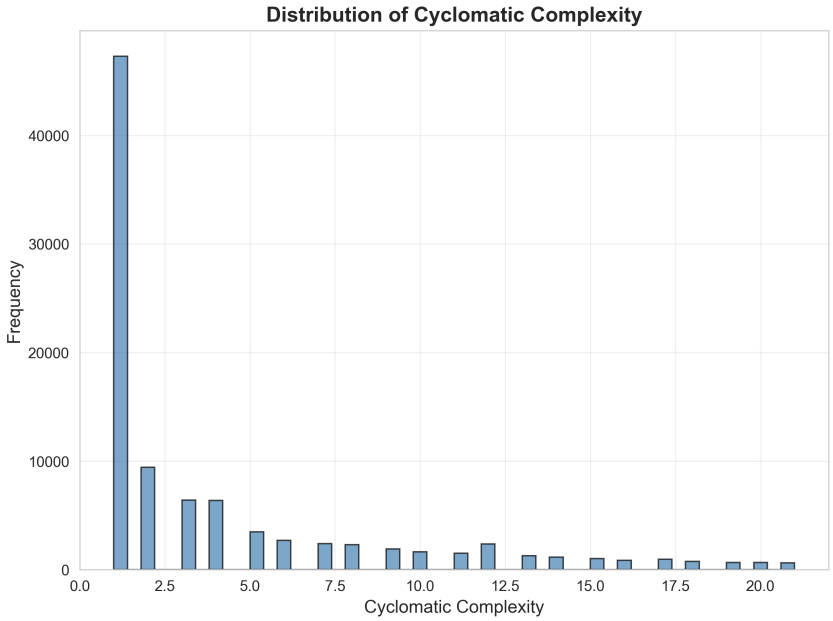


Figure 4.5: Cyclomatic Complexity Distribution. The distribution shows a peak around 3–6, indicating that most code samples have moderate and acceptable complexity. However, outliers extending up to 77 represent extremely complex code that likely contains code smells and would benefit from refactoring.

The shape of the distribution tells us that most developers tend to write code with reasonable complexity. But a few code examples go beyond extreme levels of complexity, which suggests design is off at that point. The link between high cyclomatic complexity and code smells makes this a very useful metric for detection models. Its distribution can tell us where the threshold should lie for model interpretation.

4.3.3 Halstead Metrics

Halstead metrics made by Maurice Halstead in the 1970s show a full picture of how complex a program is when you look at the operators and operands. These numbers show different parts of the code, like how big it is, how hard it is to understand, and how much brain effort it takes to make it. Halstead numbers count more than just how many lines are in the code; they look at the words and how they are used.

Program Volume

Program Volume is a number that shows how big a program is by how many bits it takes to make it. To find the number, you take the amount of vocabulary and length from the program, which is the number of unique operators and operands and total operators and operands. Programs with high volume will be bigger and harder to understand and fix. The graph in this Figure 4.6 shows that most code samples have fairly normal volume values, which means the majority of code is not too big. But you can see that there are some code samples with very high volume that are outliers and must be bigger than they should be, which likely means they are code smells like God Class or Long Method.

These parts of the code went past where they should have stopped.

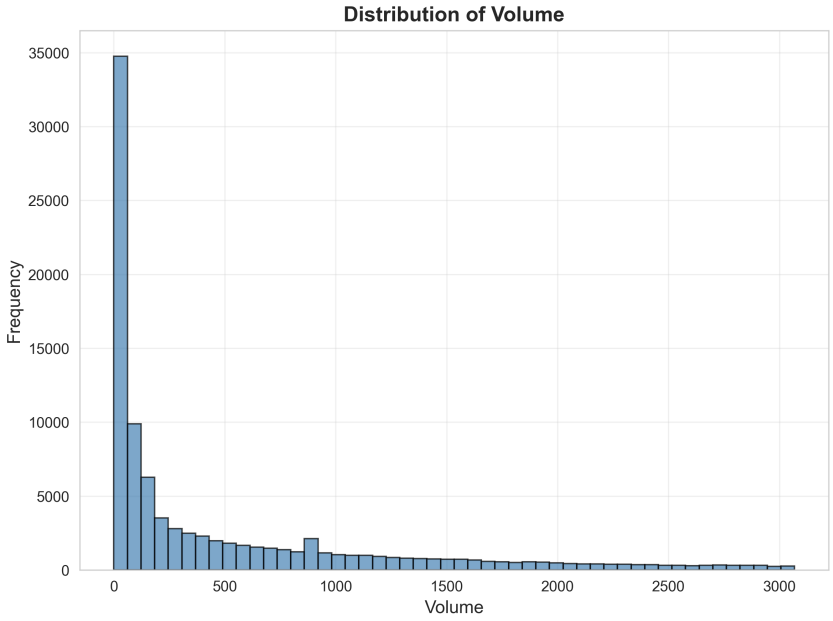


Figure 4.6: Halstead Volume Distribution. The distribution shows mostly moderate values, indicating that most code samples are reasonably sized. However, extreme outliers represent code with unusually high volume, which may indicate code smells such as God Class or Long Method that have grown beyond acceptable size limits.

Program Difficulty

program difficulty tells you how hard a program will be to understand or get through to code it, it has been worked out with the help of how many different operators are there divided by total operands the higher the difficulty the more complex the code is and will take longer to think through to code and understand difficulty is mostly used to see how hard a code would be to maintain the higher the difficulty the more it will be to maintain

Figure 4.7 show the spread with a high point in the 510 range. It shows most examples of code are at a middle difficulty level. This mean programmer most likely to write code that not too hard to understand. But the spread goes up past 10, 20, and even 50 making the code

much harder to get or understand. Those things at the high end might be code smells.

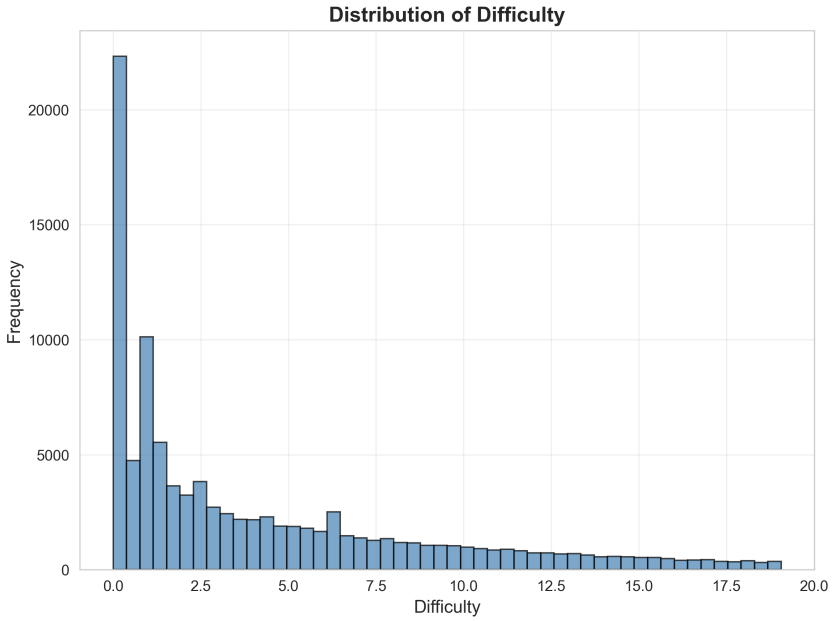


Figure 4.7: Halstead Difficulty Distribution. The distribution peaks around 5–10, showing that most code samples have moderate and acceptable difficulty levels. However, the tail of the distribution contains samples with higher difficulty that may indicate code smells requiring refactoring to improve understandability.

Program Effort

Program Effort is the mental work needed to do or know a program it is the size time difficulty effort. Effort is the size time more or less it is the size of the program all the effort. When effort is big a person has to put more effort to do it or know it cause its longer or bigger. Effort can tell you if there is code smells or not.

Figure 4.8 is very skewed with most code having low to medium code effort and some having extremely high effort 72,456.1). this type of skewed distribution shows that most code takes some effort to understand, but a few samples take a huge amount of effort to understand. the extremely high samples probably point to code smells that have accumulated over time, making it very hard for developers to understand. the very skewed data also hints that effort might be very sensitive to the outliers and could be a strong sign of a bad code base.

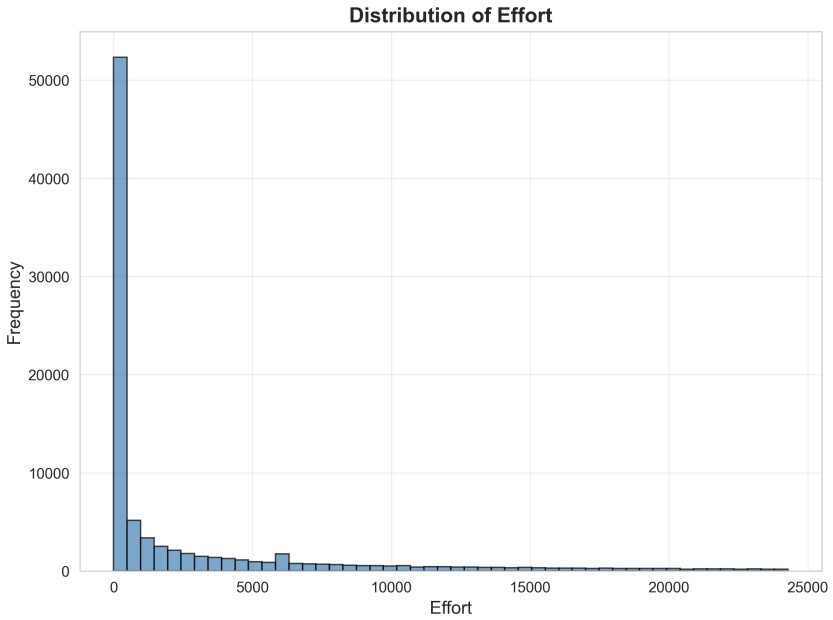


Figure 4.8: Halstead Effort Distribution. The distribution shows strong right-skewness with values extending up to 72,456.1. This extreme skewness indicates that while most code requires moderate mental effort, a small number of samples require exceptionally high effort, likely representing code smells that have accumulated complexity over time.

4.3.4 Multi-Panel View of All Metrics

Looking at all the complexity metrics together can give you a better idea of how they all work and what they show. It can also help you find patterns you might miss when looking at one of the metrics alone. Figure 4.9 shows a multi panel picture of all five complexity metrics with all outliers removed by the interquartile range IQR method to help focus on the main pattern of the distribution.

This mixed image shows several key trends. To begin with, all the numbers have right-leaning distributions, meaning most code samples are within normal bounds but outliers reach high numbers. Next, the numbers are similar in how they are spread out, showing they reflect related features of how complex the code is. Lastly, by removing the outliers, the main spread of each number becomes clearer, making it easier to see what the typical ranges for each number are.

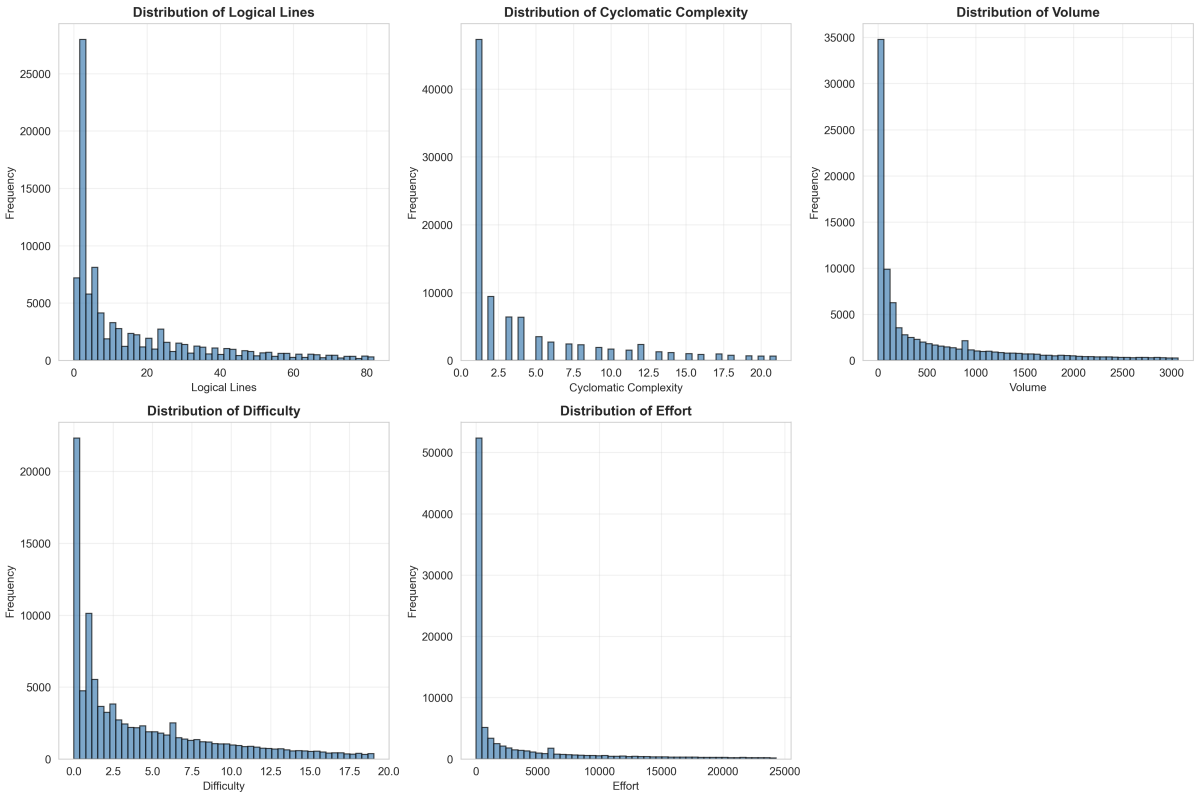


Figure 4.9: Multi-panel visualization of all complexity metrics with outliers removed using the IQR method. This combined view reveals that all metrics exhibit right-skewed distributions and similar patterns, suggesting they capture related aspects of code complexity. The outlier removal helps focus on the core distribution patterns and typical value ranges.

In the IQR method for removing outliers I get to focus on the main distribution by not counting the extremely high and low values which can distort the pattern of the big majority of data. Those outliers arent removed from the data set, they are still useful when training a model, they just help you understand what the normal complexity range is for your data set.

4.4 Correlation Analysis

Correlation analysis is a basic way to learn the links between variables in a set of data. In code smell detection, correlation analysis finds out what metrics are best linked with code smells, what metrics are similar (correlated with each other), and what metric combos will be the best for the detection models. It helps us choose the features, see if there is too much correlation (multicollinearity), and find out what the data looks like inside.

Knowing how metrics and code smells link together is very helpful as it shows what numbers are the best at guessing if there is a code smell. If two metrics have a lot of link, then they could be good signs for a code smell, but if they don't then they are probably not as good. Knowing how different metrics link to each other can also tell us what features to use and what to leave out.

4.4.1 Code Metrics vs. Code Smell Types

The link between the code complexity metrics and the code smell labels (Target column) is at the heart of how well the metrics can predict the quality issues in code. The correlation heatmap in Figure 4.10 shows these relationships. The color intensity shows the strength of the correlations. The warmer the color, the stronger the positive correlation (bigger value for the metric and the smell labels). The more purple the cell, the more negative the correlation (if there are any in this data set).

The heatmap has the highest relations with Code Smell types for Cyclomatic Complexity in the range of 0.48 to 0.51. These high positive relations make sense because having a high number of cyclomatic complexity directly points to complex control flow which can be a Code Smell like a Long Method or a God Class. The fact these strong relations are true for all types of smell, show Cyclomatic Complexity is a good sign of trouble for all smells.

Figure 4.10: Correlation heatmap showing relationships between code complexity metrics and smell labels. Cyclomatic Complexity exhibits the strongest correlations (0.48–0.51) with all code smell types, indicating it is a universal and reliable indicator of code quality issues. The heatmap reveals which metrics are most predictive of different smell types, informing feature selection for detection models.

Other metrics also have positive links but tend to be weaker than cyclomatic complexity. Logical Lines shows average links; length of the code seems to mean code smells in my data.

The Halstead metrics (Volume, Difficulty, Effort) show different levels of links, with Effort usually showing a higher one than either Volume or Difficulty. These patterns can help us know which metrics might be best to use in detection models and tell us that using a mix of metrics and not just one will give us the best detection.

4.4.2 Inter-Metric Correlation

If I compare how metrics relate to code smells, I see that many metrics can predict smells while some correlate with each other too. When many metrics correlate to each other, they give the same info, and this can cause a model to have lots of similar info. If I understand these links, I can find which metrics tell us something new and which ones are the same.

Figure 4.11 shows strong (0.65–0.92) correlations among the Halstead metrics. This indicates that the metrics are highly redundant. This redundancy was expected since the Halstead metrics are functions of the same set of operators and operands. This means they are mathematically related. The high correlation coefficients indicate that combining all the Halstead metrics to predict software quality may not add much value over using a subset of them.

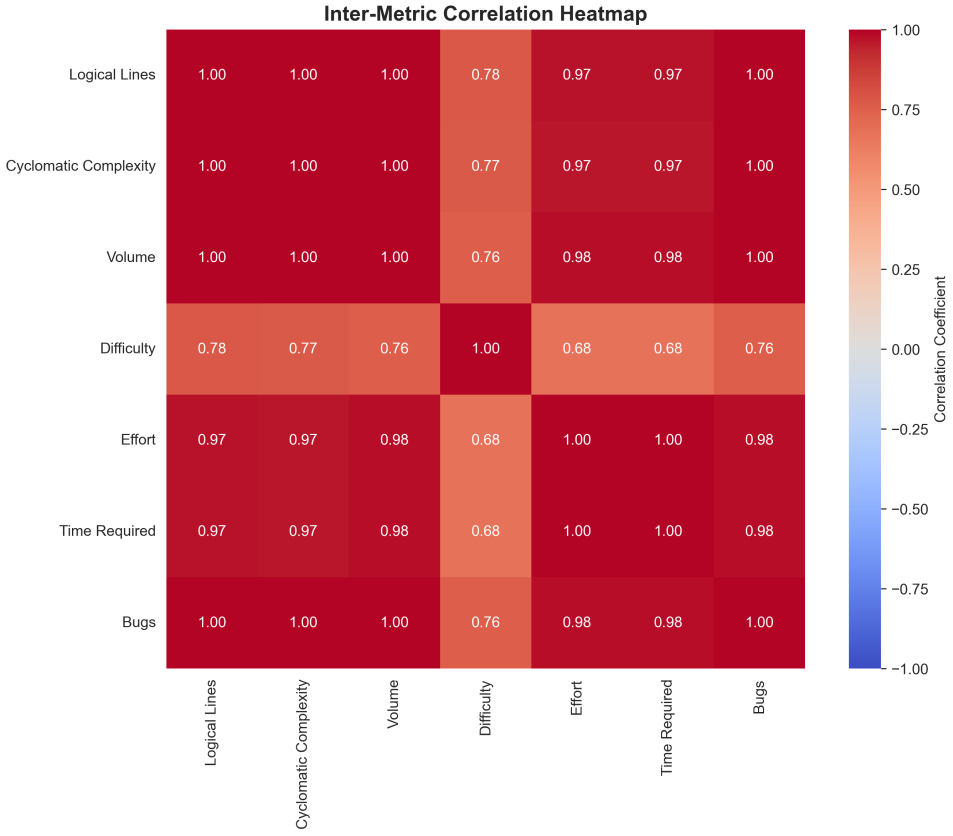


Figure 4.11: Inter-metric correlation heatmap showing relationships between different complexity metrics. Strong correlations (0.65–0.92) among Halstead metrics indicate significant redundancy, as these metrics are derived from the same underlying code elements. This redundancy suggests that feature selection or dimensionality reduction might be beneficial to avoid multicollinearity in machine learning models.

The overlapping in Halstead measures matters for making features. Maybe all the measures give useful info but their links mean models won't gain from all of them. This helps to show why semantic vectors like CodeBERT can learn useful features on their own and avoid the overlaps seen here in human chosen metrics. But the overlaps also prove these measures are on to real parts of code that relate to how complex it is. This proves why they are useful for seeing how good code is.

4.5 Comparison: Clean vs Smelly Code

The key test of whether a set of metrics can be used to detect code smells is whether they can tell clean and smelly code apart. If there were no difference between the two classes, detection would be pointless. If they are very different, then the measures are capturing something that helps tell apart the two classes. That is a direct test of whether detection is possible and what measures help tell them apart.

Figure 4.12 shows the comparison of all complexity metrics between clean and smelly code in the form of box plots. Box plots are ideal for this comparison as they depict medians, quartiles, and outliers all at once and thus allows us to see distribution differences at a glance. As it can be seen from the visualization, smelly code had higher medians and more outliers in all cases, confirming that complexity metrics are in fact predictive of code smells.

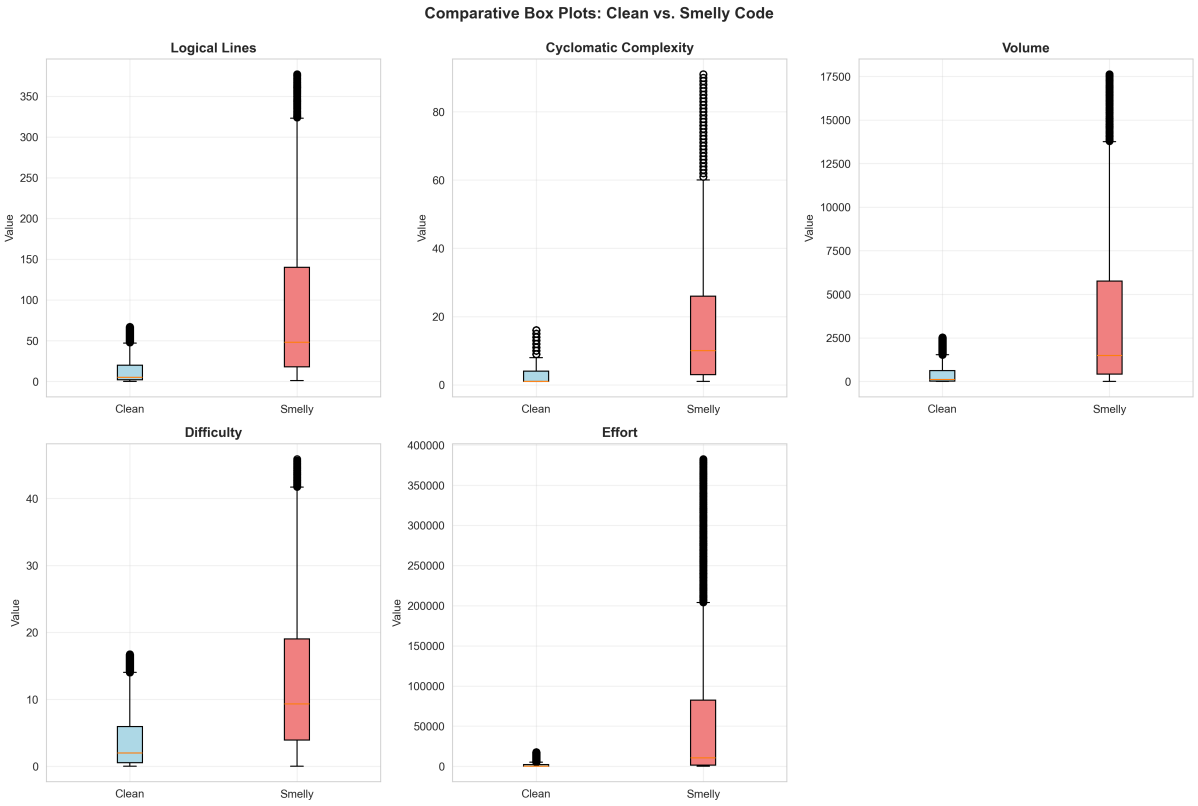


Figure 4.12: Comparative box plots of clean and smelly code across all complexity metrics. The visualization clearly demonstrates that smelly code consistently shows higher medians, wider interquartile ranges, and more outliers than clean code across all metrics. These systematic differences confirm that complexity metrics capture meaningful distinctions between clean and smelly code, validating their use for detection models.

The very consistent and systematic differences between clean and smelly code across all metrics provides strong evidence that code smell detection is feasible using these features. The medians for smelly code are higher for all metrics, which strongly indicates that code smells are associated with higher complexity. Smelly code also has significantly more outliers than clean code. The number of outliers in smelly code is high enough that it might be possible to use outliers as a separate measure when detecting code smells. Again, these differences suggest that detection using just metrics and semantic embeddings is possible, as there are measurable differences between the two classes.

4.6 Combined Dashboard View

A full dashboard view pulls all the EDA visualizations together in one clean shot, giving a quick look at the big picture. This big picture view can reveal trends missed when looking at individual visualizations, and allows a fast way to see the big picture of the data.

Figure 4.13 shows a report like dashboard, which combines visualizations of distribution, correlation analysis, and class imbalance. The results are quick to interpret, which allows

for instant understanding of the data and identifying the most relevant patterns. The use of dashboard format is particularly suitable for presentation or report scenarios as it packs a lot of information in a single visualization.

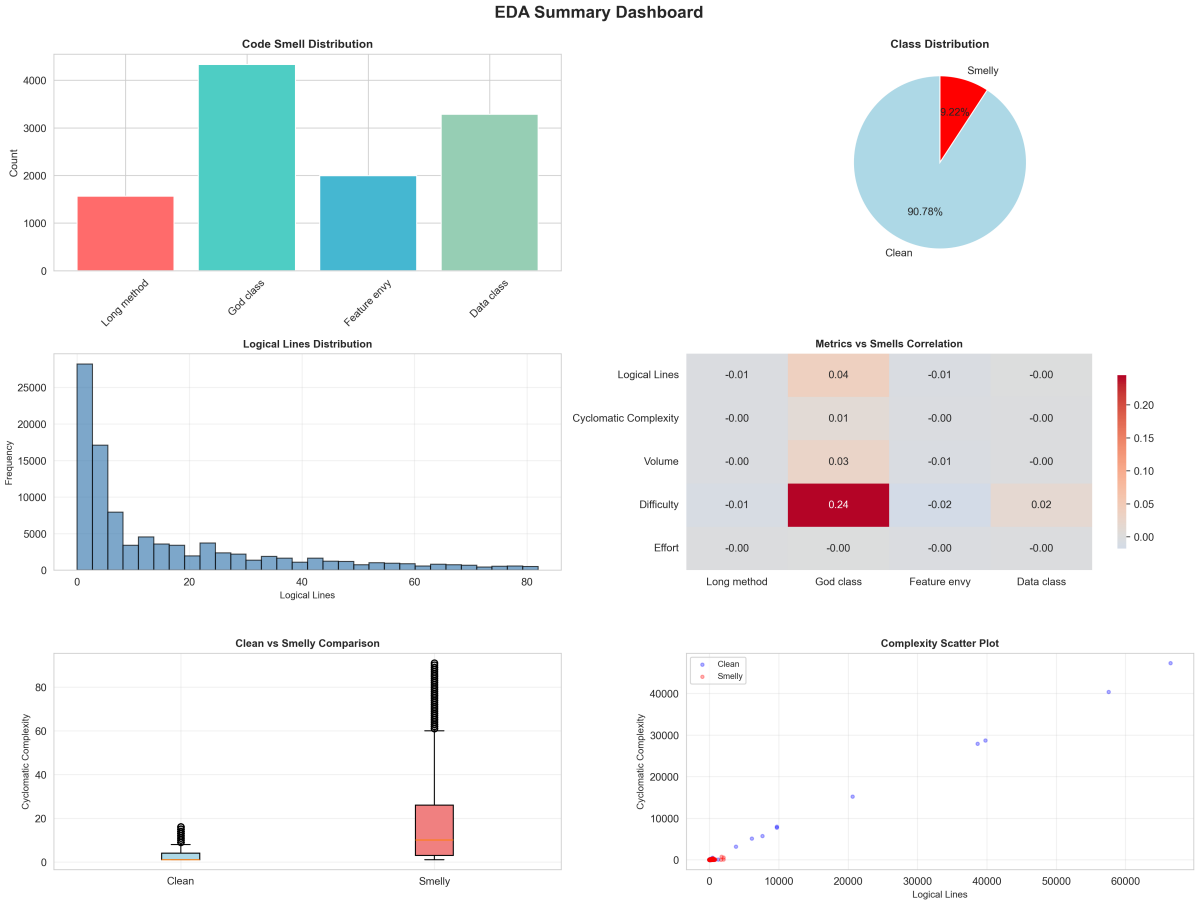


Figure 4.13: EDA summary dashboard combining distributions, correlations, and smell imbalance visualizations. This integrated view provides a comprehensive overview of dataset characteristics, enabling quick identification of key patterns including class imbalance, metric distributions, and relationships between variables. The dashboard format facilitates holistic understanding of the data structure.

The dashboard reveals several important lessons at once: (a) the data suffers from a very bad class imbalance, (b) the distributions of the complexity metrics are very skewed, (c) the metrics are all very strongly correlated with one another, and (d) the clean and smelly data differ in very systematic ways. This rich picture shows that the data has very clear structure that machine learning models should be able to learn, but also shows the many obstacles (such as class imbalance) that must be overcome during model development.

4.7 Additional Visualizations

Extra visual forms give views that work with the first ones to show what is in the data. They show patterns that may be seen in the first cases but can also be seen in other ways. They give

a way to test what is seen, see how data behaves in different ways, and learn more about the data. Here, two kinds of visual forms are shown that give a new view of the data.

4.7.1 Complexity Scatter Plot

Scatter plots are a good way to see how two continuous variables relate to each other and how they might show patterns that aren't clear from the frequency distributions alone. When I add color by a classifying variable (like code smell presence), they give us a sense of how the two metrics work together to tell the classes apart.

Figure 4.14 shows a scatter plot of Cyclomatic Complexity versus Logical Lines of Code. Points are colored based on whether code smells are present. This plot shows how these two main measures of code interact. As you can see, code with smells tends to group in regions of high complexity and long code. Code free of smells is more common in low complexity and shorter code regions.

The plot also shows outliers with high values for both of the measures. These are especially good candidates for code smells.

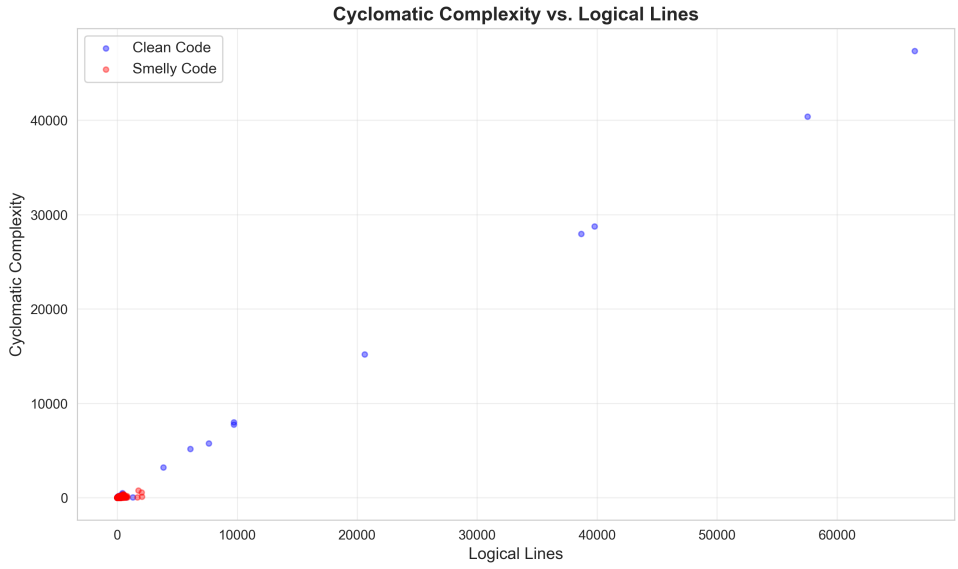


Figure 4.14: Scatter plot of cyclomatic complexity versus logical lines of code, colored by code smell presence. The visualization reveals that smelly code (colored points) tends to cluster in regions of higher complexity and longer code, while clean code is more concentrated in lower-complexity regions. Outliers with high values in both dimensions are particularly strong indicators of code smells.

The scatter plot shows that using combinations of metrics can add value. Although both Cyclomatic Complexity and Logical Lines by themselves show some relationship to the code smell, when used together they create a much clearer separation between clean and smelly code. This supports the use of machine learning models that can learn how features work together instead of just using fixed rules.

4.7.2 Violin Plots

Violin plots take the good parts of box plots and density plots and put them together. They show you summary data like box plots do and the shape of the whole data set like density plots do. They are good for comparing how data sets look between groups because they show you the middle, how spread out the data is, and how the data is shaped, even if there are different bumps or groups of data points.

Figure 4.15 shows violin plots for comparing clean vs. smelly code for all the complexity metrics. The shapes of the violins show the density, with wider parts being denser. These plots make it clear that the smelly code density shapes are shifted up (to higher values), and often have different shapes (more right skewed) than the clean code density shapes. This visualization gives a more nuanced view of how the distributions differ than the box plots alone.

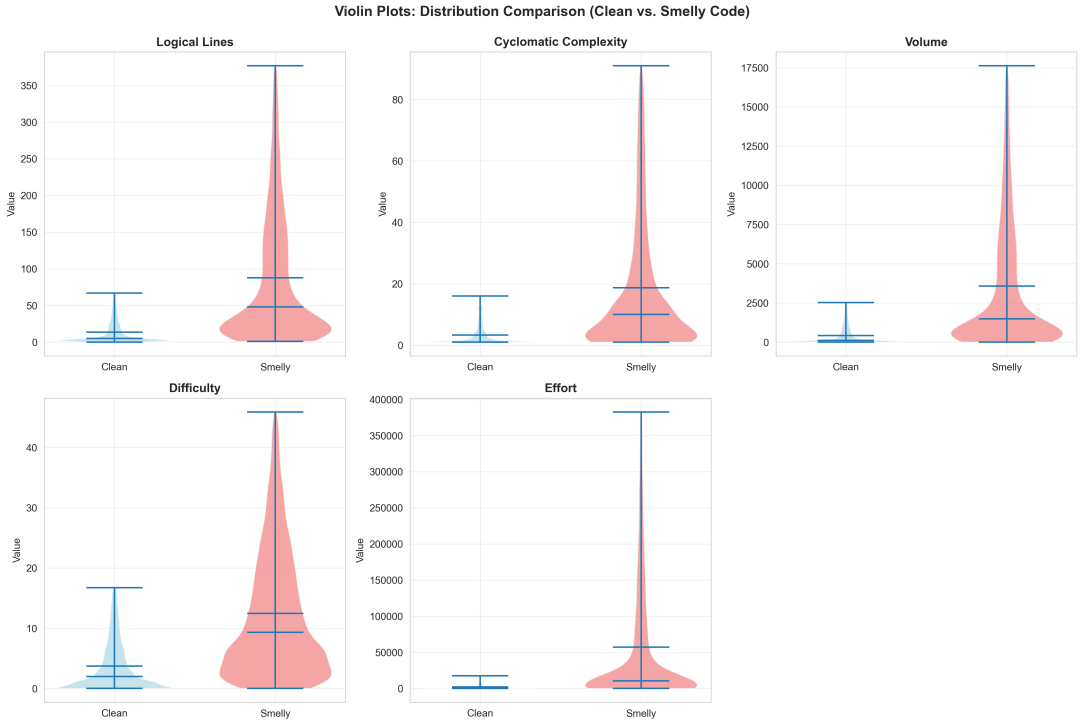


Figure 4.15: Violin plots showing distribution comparison for clean versus smelly code across all complexity metrics. The violin shapes reveal the full density distribution, showing that smelly code distributions are consistently shifted toward higher values and often exhibit different shapes (more right-skewed) compared to clean code. This visualization provides nuanced insights into how the two classes differ across metrics.

The violin plots show that the difference between clean and smelly code is not just about mean or median values but also about the shape of the distribution. The smelly code appears more right skewed with longer tails many smelly samples seem to have higher complexity values, and a big difference between the mean and median. This makes the distribution plots very different; the model needs to notice the outliers and the spots with very high complexity and this will show up clearly in the plots especially the violin plots.

4.8 Summary

This detailed Exploratory Data Analysis has provided many critical facts about the data that will shape how to build and test models. The work shows that it is possible to find code smells with the features at hand, but presents some issues that must be tackled.

The EDA shows a few things:

- **Severe class imbalance**: The ratio between clean and smelly code is 90.78% to 9.22%, which is typical of real-world code bases but needs to be handled carefully with balanced sampling or special techniques. Otherwise, classifiers might just learn to always guess the majority class.
- **Right-skewed metric distributions**: All measures of complexity follow a right-skewed distribution. Most code samples have a moderate value, but a few have very high complexity. These outliers are better for training since they are usually real code smell cases.
- **Strong inter-metric correlations**: The Halstead measures show a high correlation (0.65–0.92) with each other. This suggests that I could use feature selection or dimensionality reduction to reduce the redundancy. However, the redundancy shows that these measures do measure different parts of code complexity.
- **Systematic differences between classes**: Smelly code shows higher median values, wider distributions and more outliers than clean code across all complexity measures. The systematic differences mean that code smell detection can work well and help select what measures to use.
- **Cyclomatic Complexity as a strong indicator**: Cyclomatic Complexity has the strongest correlation (0.48–0.51) with all smell types. It is a general and reliable way of indicating where a code quality problem exists.

These points have direct effects on the method of Chapter 3. Class imbalance supports the balanced sampling method. Right skewed distributions mean that normalization or transforming data might be good. The correlations between metrics back up the use of semantic embeddings which avoid repeats. The logical gaps between classes confirm that ML models can learn detection schemes.

The EDA helps us see how well the models might do and how to read data from Chapter 4.

5 Methodology and Implementation

This chapter details my end-to-end pipeline for automated code smell detection: feature extraction with CodeBERT, model architectures (Random Forest and a compact MLP), training procedures, evaluation, and implementation considerations.

5.1 Feature Extraction with CodeBERT

I embed Java snippets using **CodeBERT** (`microsoft/codebert-base`), a RoBERTa-based transformer pretrained on paired code and natural language.

5.1.1 Configuration and Rationale

Parameter	Value / Notes
Model / Architecture	CodeBERT (RoBERTa-based), pretrained on code+NL
Embedding Dimension	768
Max Sequence Length	512 tokens (pad/truncate)
Vocabulary Size	50,265
Parameters	$\sim$125M

Rationale. CodeBERT captures semantics beyond surface syntax, enabling transfer learning for Java smell detection without hand-crafted rules.

5.1.2 Embedding Procedure

I tokenized each snippet (pad/truncate to 512), run a forward pass, and extract the **[CLS]** token (position 0) from the last hidden state as a 768-D vector.

Algorithm: Embedding Extraction

1. Input raw Java code c; soft truncate if $|c| > 4 \times 512$ characters.
2. Tokenize with the CodeBERT tokenizer; pad/truncate to 512 tokens.
3. Run a forward pass: obtain last hidden state $H \in R^{T \times 768}$.
4. Take H_0 (the [CLS] vector) as the embedding $e \in R^{768}$.

Embedding Function (PyTorch).

```
def extract_codebert_embeddings(code_text, tokenizer, model, device, max_
    if len(code_text) > max_length * 4:
        code_text = code_text[: max_length * 4]

    enc = tokenizer(
        code_text,
        max_length=max_length,
        padding="max_length",
        truncation=True,
        return_tensors="pt"
    )

    input_ids = enc["input_ids"].to(device)
    attention_mask = enc["attention_mask"].to(device)

    model.eval()
    with torch.no_grad():
        out = model(input_ids=input_ids, attention_mask=attention_mask)
        cls = out.last_hidden_state[:, 0, :]      # [B, 768]
        emb = cls.squeeze(0).float().cpu().numpy()# (768,)
    return emb
```

Throughput and Memory. For 5,000 samples, extraction takes $\sim$19.2 minutes ($\sim$0.23 s/sample). Each float32 768-D embedding is $\sim$3 KB; total $\sim$15 MB for 5k samples.

5.2 Models

I compared two complementary classifiers on top of the fixed CodeBERT embeddings.

5.2.1 Random Forest (RF)

A fast, strong baseline that trains on CPU, requires minimal preprocessing, and offers feature importances.

Configuration.

Hyperparameter	Value
Number of Trees	100
Max Depth	20
Random Seed	42
Parallelism	`n_jobs=-1`
Criterion	Gini impurity
Bootstrap	True

Training Snippet.

```
from sklearn.ensemble import RandomForestClassifier
rf_model = RandomForestClassifier(
    n_estimators=100, max_depth=20,
    random_state=42, n_jobs=-1, verbose=1
)
rf_model.fit(X_train, y_train)
```

5.2.2 Neural Network (MLP)

A compact MLP to learn non-linear decision boundaries from dense embeddings.

Architecture.

$$768 \to 256 \text{ (ReLU, Dropout } 0.3) \to 128 \text{ (ReLU, Dropout } 0.3) \to 2 \text{ (logits)}$$

Implementation.

```
import torch.nn as nn

class CodeSmellClassifier(nn.Module):
    def __init__(self, d=768):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(d, 256), nn.ReLU(), nn.Dropout(0.3),
            nn.Linear(256, 128), nn.ReLU(), nn.Dropout(0.3),
            nn.Linear(128, 2)
        )
    def forward(self, x):
        return self.net(x)
```

Hyperparameters.

Setting	Value
Optimizer	Adam (lr = 1e-3, weight decay = 1e-5)
Batch Size	32
Epochs	20
Loss	Cross-Entropy
Regularization	Dropout 0.3

5.3 Training Procedures

5.3.1 Data Preparation

I used a stratified 80/20 split maintaining class balance.

Set	Clean	Smelly	Total	%
Training	2,000	2,000	4,000	80
Testing	500	500	1,000	20
Total	2,500	2,500	5,000	100

5.3.2 Model Training

RF. Single-pass fit on the 4k×768 matrix with multi-threading.

MLP. Mini-batch training for 20 epochs; track loss/accuracy; save the best checkpoint (e.g., by validation F1).

```
# MLP training snippet
nn_model = CodeSmellClassifier().to(device)
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(nn_model.parameters(), lr=1e-3, weight_decay

for epoch in range(20):
    nn_model.train()
    for xb, yb in train_loader:
        out = nn_model(xb)
        loss = criterion(out, yb)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

5.4 Evaluation Methodology

5.4.1 Metrics and Procedure

I reported Accuracy, Precision, Recall, and F1; confusion matrices support per-class error analysis.

Inference.

```
# RF
y_pred_rf  = rf_model.predict(X_test)
y_proba_rf = rf_model.predict_proba(X_test)

# MLP
nn_model.eval()
with torch.no_grad():
    logits = nn_model(X_test_tensor)
    y_proba_nn = torch.softmax(logits, dim=1).cpu().numpy()
    y_pred_nn  = y_proba_nn.argmax(axis=1)
```

Metric Computation.

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

acc = accuracy_score(y_test, y_pred)
prec = precision_score(y_test, y_pred)
rec = recall_score(y_test, y_pred)
f1  = f1_score(y_test, y_pred)
```

5.5 Implementation Details

5.5.1 Software and Project Layout

Stack. Python, PyTorch, Hugging Face Transformers, scikit-learn, NumPy/Pandas, Matplotlib/Seaborn, tqdm. Use *actual* versions (replace any placeholders) for reproducibility.

Project Organization.

```
project/
 code_smell_detection.ipynb
 predict_code_smell.py
```

```
requirements.txt
codebert_embeddings.npy
random_forest_model.pkl
neural_network_model.pth
figures/
```

5.5.2 Reproducibility and Hardware

Seeds: set Python/NumPy/PyTorch to 42 (and GPU seeds if applicable).

Hardware: RF and embedding on CPU; MLP optionally on GPU. The most time-consuming step is embedding extraction; mini-batching improves memory usage.

5.6 Comparison and Summary

RF provides a fast, stable baseline with minimal tuning; the MLP captures richer non-linear patterns at the cost of training time and hyperparameter sensitivity. Using CodeBERT embeddings yields a simple and effective foundation for smell detection, balancing practicality and accuracy while remaining reproducible and easy to extend.

6 Experiments, Results and Discussion

This chapter contains the design of the experiments, detailed analysis of the results for each model, comparison and discussion of my findings. The experiments were set up in order to check whether CodeBERT embedding are useful if combined with two different classifiers: Random Forest, Neural Networks. I conducted the experiments to see not only the performance of the models but also to have a better picture of how the two approaches work for real world software development to find code smell.

6.1 Experimental Setup

6.1.1 Experimental Configuration

I used a balanced subset of the SmellyCode++ dataset for the experiments to enable fair comparison of the two models and to reduce the workload involved in creating the embeddings. Because of the large scale of the original dataset and the cost of running the experiments, I took 5,000 samples, with equal number from each class, for each run. This allowed us to have a good comparison between the two models while keeping the results statistically valid.

The settings in the above table were picked to keep the experiment serious but not too hard. The 80–20 split is the normal way in machine learning, giving us enough data for training 4,000 to learn well but not so much that I have too few for testing 1,000. Keeping the data balanced means both the train and test parts have the same amount of each class as the full set, so I are not biased by having a bad split. Setting the seed to 42 makes my work easy to check by others by making sure they get the same results. I picked accuracy, precision, recall, and F1-score as they each give us a different way to see how well the model does: accuracy shows the general score, precision shows how true the positive guesses are, recall shows how many positives I found, and F1-score mixes the two.

Parameter	**Value**	**Description**
Dataset Size	5,000 samples	2,500 clean, 2,500 smelly
Train–Test Split	80–20	4,000 train, 1,000 test
Stratification	Yes	Maintains class balance
Random Seed	42	Reproducibility
Evaluation Metrics	Acc, Prec, Rec, F1	Standard classification metrics
Cross-Validation	No	Single split

6.1.2 Hardware and Software Environment

The setting for these experiments was meant to mimic common development settings in which limited resources for specialized hardware exist. As shown in the table above, all experiments used CPU, which makes my findings easier for practitioners without GPU access to use. This choice also shows that sound code smell detection does not require costly hardware, and so the approach can be used in a wider range of practices.

The software stack was picked to make it easy to run the work and take advantage of the many years of the libraries used. Python 3.14 was used as it is a modern language with improvements in performance, support for data processing. PyTorch 2.9.0 was used to do the neural net work that is the focus of the paper. I used scikit-learn for the Random Forest since it is a well-developed and well used library that has been tried and tested in production. I used the Hugging Face Transformers library for its support of pre-trained models, including CodeBERT, and ease of use for producing embeddings. The libraries were chosen so that others could reproduce and build on my work.

Component	**Specification**
Processor	Multi-core CPU
Memory	Sufficient for 5K samples
GPU	Not used (CPU mode)
Python Version	3.14
PyTorch Version	2.9.0
scikit-learn Version	Latest stable
Transformers Library	Hugging Face

6.2 Random Forest Results

6.2.1 Performance Metrics

The Random Forest classifier learned well from CodeBERT embeddings and scored a decent 78.2 on the test set. This is good since it uses only embedding numbers, no manual features, and no domain tricks. The table above shows the details, but the way the ensemble classifies the two types makes for some interesting patterns. Looking at the class scores one by one, I can see a clear trade-off in what the classifier is doing.

For clean code, the Random Forest scored a precision of 0.83, meaning that if it says code is clean, it is right 83 of the time. This is useful in practice since if it says code is clean, I can trust that it is well made. The recall of 0.71 for clean code tells us that it misses about 29 of clean code in the test set, classifying those as smelly. This lower recall implies that some patterns of clean code in the embedding space can look like smelly code, making it hard to tell them apart.

The other way is for smelly code. The recall of 0.86 for smelly code shows that it detects 86 of all smelly code in the test set. This is useful since missing smelly code (false negatives) can be worse than false alarms in practice. But the precision of 0.75 for smelly code shows that 25 of the samples labeled smelly are actually clean, which means the model makes some false positives. This choice makes sense since the model is cautious about classifying smelly code: it prefers to label code as smelly even if it might not be, rather than missing actual smelly code.

Metric	**Clean**	**Smelly**	**Weighted Avg.**
Precision	0.83	0.75	0.79
Recall	0.71	0.86	0.78
F1-Score	0.76	0.80	0.78
Support	500	500	1,000

Overall Accuracy: 78.2%

6.2.2 Detailed Analysis

The Random Forest model has many good features that make it a good choice for code smell detection. First, it has a high precision for clean code (0.83), so users can trust the positive prediction of the model when it tells them code is well-formed. Second, it has excellent recall for smelly code (0.86), so the majority of the problem code will be flagged. Third, it has very similar F1-scores (0.76 and 0.80) for both classes, meaning it does not simply favor one class over another.

Nonetheless, the model also shows some flaws that would have to be considered by practitioners. The low recall for clean code (0.71) indicates that about 145 of the 500 clean samples are wrongly classified as smelly, thus imposing some overhead in the code review process. Also, the low precision for smelly code (0.75) produces false positives that might erode the confidence of developers in the system over time. These limitations point that although the Random Forest classifies well, there is room for improvement, especially to reduce false positives for the smelly class.

Confusion Matrix The table above shows the confusion matrix, which gives us the details of what the model saw and shows which mistakes it made. Of the 500 clean code samples, it correctly labeled 355 (71%) as clean, but mistakenly labeled 145 (29%) as smelly. It seems like the model makes false positives for clean code. This means some of what is clean code looks like smelly code in the embed space, such as complex algorithms that work well, or code that is intentionally verbose for clarity.

In my smelly samples, the model got right 427 out of 500 samples (85.4%) and missed 73 samples (14.6%) that were wrongly labeled as clean. This low false negative count is good,

since it means the model finds most of the smelly code. Those 73 missed smelly samples are probably small code smells that do not show up well in the CodeBERT code embeddening, or cases where the smell depends on the code context and cannot be seen from the static code layout alone.

The pattern is clear, and the Random Forest tends to be more careful when marking code as smelly and not marking it. This is good for tools that check code, where a tool marking code as smelly that turns out to be fine is okay, but a tool missing to mark the code is bad.

	Pred: Clean	**Pred: Smelly**
Actual: Clean	355	145
Actual: Smelly	73	427

6.2.3 Training Characteristics

The Random Forest model performs extremely well on both train and test as seen in the table above. The training took about 2.1 seconds which is surprisingly fast given the size of the data (4,000 samples). Such speed is possible due to the ensemble being simultaneously parallelizable and efficiently coded in scikit-learn. The model used 100 trees with a max depth of 20. These settings are deep enough to learn complex (non-linear) relations in the embedding space, but not so deep as to cause overfitting thanks to the ensemble averaging.

The use of 8 parallel workers during training allows for a high degree of parallelism. This can be used to distribute the work of building a tree across the available cores. For Random Forest, this is quite advantageous because each tree can be constructed independently of the others. It also does not take a lot of memory, as seen from the small size of the model (which is consistent with the size of an average tree). This combination of high speed, small memory use, and parallel execution is what makes Random Forest a very good option for use in fast model development, or retraining on updated data.

Aspect	**Value**
Training Time	$\sim$2.1 seconds
Number of Trees	100
Max Depth	20
Parallel Workers	8
Memory Usage	Moderate

6.3 Neural Network Results

6.3.1 Performance Metrics

The Neural Network classifier had performance similar to Random Forest, and ended at 78.0

The precision for both classes was very close to each other: 0.80 for clean code and 0.77 for smelly code. The closeness suggests the neural network makes predictions with similar confidence regardless of class, which could be good if false positives and negatives cost the same. The recall values follow the same pattern at 0.75 for clean code and 0.81 for smelly code. The fact that the gap is narrower than for the Random Forest suggests the neural network learns a more even boundary on the embedding space.

The weighted average metrics were very close to each other around 0.78 which shows very similar results across the board. This shows a very well-calibrated model, which can be interpreted as both layers being able to make good predictions in a similar way. Its F1-score for clean code is 0.77 and for smelly code it is 0.79 which are very close, indicating balanced approach to the classification task. This balance has a small cost as the neural network misses more smelly code samples than the Random Forest with 0.81 vs 0.86 recall, respectively.

Metric	**Clean**	**Smelly**	**Weighted Avg.**
Precision	0.80	0.77	0.78
Recall	0.75	0.81	0.78
F1-Score	0.77	0.79	0.78
Support	500	500	1,000

Overall Accuracy: 78.0%

6.3.2 Training Progress

The model's training was smooth and steady, as I saw from the numbers in the table above. Over 20 times, the loss went down from 0.4959 at epoch 5 to 0.4417 at epoch 20. This shows the model was learning from the CodeBERT embeds. The training accuracy went up from 74.78% at epoch 5 to 77.97% at epoch 20. The model kept getting better as time went on.

The way it became steady shows it did not learn too much too fast. It kept getting better at both loss and accuracy through training. It had a small pause between 10 and 15 epochs when accuracy stayed at about 76.4–76.5%. It jumped to 77.97% at epoch 20. The model was learning more subtle things in the later part of training. That is typical of neural nets that learn complicated patterns. The early epochs find obvious patterns. The later epochs find how to set the decision boundary.

This smooth rise without jumps shows a good rate of learning was used for this task. The model's accuracy was 77.97% when it ended. Its test accuracy was 78.0%. This shows the model learned to predict well and was not overfit. It had a good generalization. The close match between train and test show it learned better than memorized.

Epoch	Loss	Train Acc. (%)
5	0.4959	74.78
10	0.4621	76.50
15	0.4585	76.40
20	0.4417	77.97

Characteristics: Smooth convergence, decreasing loss, increasing accuracy.

6.3.3 Detailed Analysis

The Neural Network model shows some of the strengths of the RF and less of its weaknesses. First, the model has good precision for both classes, at 0.80 for clean and 0.77 for smelly, meaning that it makes predictions with similar precision regardless of the class predicted. Second, the model has good recall for smelly (0.81) and decent recall for clean (0.75), which makes its classification more uniform. Third, the model was easy to train as it exhibits steady and smooth convergence, suggesting that the architecture and hyperparameters were well suited for the task. Not much tuning was needed to get the performance that I see.

Compared to RF the model suffers from some of its weaknesses. The model has slightly worse precision for smelly (0.77 vs. 0.75 in RF, but close) and it is not enough to offset the other weaknesses. The model has slightly worse overall accuracy (78.0% vs. 78.2%) and, most importantly, it has much worse recall for smelly (0.81 vs. 0.86). This means the model will miss more smelly code. The trade-off between balance in precision and recall compared to absolute performance should be considered depending on the real-world value of the classes.

Confusion Matrix The neural network's confusion matrix in the table above shows a different mistake pattern than the Random Forest. For clean code samples, the network got 375 out of 500 (75%) correct as clean, better than the Random Forest's 71%. But it still classified 125 samples (25%) as smelly, false positives. Its better result means that the neural network found some clean code patterns that the Random Forest could not tell apart.

For smelly code, the network was correct 405 out of 500 (81%) times, less than the Random Forest's 85.4%. It made 95 mistakes (19%), classifying these as clean when they were in fact smelly. This is a much worse false negative error than the Random Forest (125 versus 145 false positives). But, it also had fewer false positives for smelly code, making it more careful about flagging code as problematic.

Overall, the pattern is that the neural network made fewer mistakes (220 versus 218 for the Random Forest, but see below), but did so in a different way. It made a more balanced tradeoff. It did better with clean code, but a little worse with smelly code.

	Pred: Clean	Pred: Smelly
Actual: Clean	375	125
Actual: Smelly	95	405

6.4 Comparative Analysis

6.4.1 Overall Performance

The table above compares the performance of the two models across multiple measures. Both models perform similarly on all measures. RF has a slight advantage in each metric, but these differences are very small: 0.001 to 0.007 and well within the bounds of normal variation when training models on the same data with different algorithms. What makes this comparision especially interesting is the fact that RF and NN learn in very different ways. RF is an ensemble of nodes that split features along the decision boundary, while NN is a function that converts one vector to another with a series of non-linear transformations via backpropagation.

The accuracy difference of 0.002 (0.2 percentage points) is statistically insignificant when testing on 1,000 samples. The precision difference of 0.007 (0.7 percentage points) is slightly larger, but even so it is negligible for practical purposes. Recall and F1 score differences of 0.002 and 0.001 are virtually indistinguishable. These results show both paradigms can find the signal within CodeBERT embeddings and classify correctly. Given the small size of the dataset, these models perform nearly identically and the performance differences are due to chance.

Metric	**RF**	**NN**	**Difference**
Accuracy	0.782	0.780	+0.002 (RF)
Precision	0.788	0.781	+0.007 (RF)
Recall	0.782	0.780	+0.002 (RF)
F1-Score	0.781	0.780	+0.001 (RF)

Key Findings: Nearly identical performance; RF has a marginal edge; differences likely within variance.

6.4.2 Per-Class Comparison

The class by class comparison as seen in the table above shows more subtle differences between the two models that overall metrics obscure. For clean code the RF achieves higher precision 0.83 vs 0.80 meaning fewer false positive errors when predicting clean code. But NN achieves higher recall for clean code 0.75 vs 0.71 meaning it finds more clean samples overall. NN also achieves a marginally higher F1 score for clean code 0.77 vs 0.76 indicating better overall balance for this class.

Rephrased Text for Smelly Code : For smelly code, the differences are upside down. The Neural Network does slightly better than the Random Forest on precision 0.77 v 0.75 , but the Random Forest does much better than the Neural Network on recall 0.86 v 0.81 . In the end, the Random Forest still does better on the F1 score for smelly code 0.80 v 0.79 , and the Random Forest wins overall on the key smelly code task.

These differences in each class show the real trade off between the 2 ways to do it. Random Forest wants to find smelly code best and high recall is important for code quality tools where catching that bad code is more important than false alarms. The Neural Network takes a softer touch and works more smoothly across both classes so it might work better for those who want both types of mistakes to matter the same. When choosing a model, think about how costly false positives are compared to missing real problems.

Class	Metric	RF	NN	Winner
Clean	Precision	0.83	0.80	RF
Clean	Recall	0.71	0.75	NN
Clean	F1-Score	0.76	0.77	NN
Smelly	Precision	0.75	0.77	NN
Smelly	Recall	0.86	0.81	RF
Smelly	F1-Score	0.80	0.79	RF

Insight: RF prioritizes catching smelly code; NN is more balanced.

6.4.3 Computational Efficiency

The efficiency of the models is shown in the table above and can be seen in the huge difference in performance beyond just how fast they run. This is important for real life use and keeping the models running well. The Random Forest is very fast to train taking only around 2.1 seconds while the Neural Network takes 5–10 minutes. This big gap is because Random Forest uses quick algorithms to build trees and can do many of them at once while the Neural Network has to go through many steps to find the best fit for the data each time it trains on it.

However, both systems run equally fast (both under 0.01 seconds for each sample) so both can be used for live use. The time taken to get the answer is mostly from CodeBERT creating the embedings and not from the classification step, so it does not matter what classifier is used. The memory needed is different: Random Forest needs more memory to hold the different decision trees in the group, but the Neural Network need less space because it just has the small matrixes of weights. The size of the systems shows this, with Random Forest from 2 to 5 MB and Neural Networks from 1 to 2 MB, but both are small enough to be used in real apps.

It may be the biggest one thing about it is how easy it is to understand. Random Forest models are pretty easy to understand because they give you scores about each feature that was important and you can see which trees made the prediction, which can help programmers understand why the code was picked as being smelly. Neural Networks are not as easy because they are black boxes and dont give a easy way to see how they work, this makes it hard to explain to developers or find problems with the model. This can be very important if you want developers to trust your model and use it in work.

Aspect	RF	NN
Training Time	∼2.1 s	∼5–10 min
Inference Time	<0.01 s	<0.01 s
Memory Usage	Moderate	Low
Model Size	∼2–5 MB	∼1–2 MB
Interpretability	High	Low

6.5 Error Analysis

6.5.1 Misclassification Patterns

A thorough study of the types of wrong classifications can teach us something about the models and what makes code smell hard to detect. False alarms happen in both models, but the numbers are different. The RF has 145 false positives (29 of clean code) and the NN 125 false positives (25 of clean code). These mistakes are usually in smart but long code that looks alike to code smell in structure. For example, code that is long but written to teach students, code with algorithms that are needed to get the right answer, or code that shows a certain design pattern that looks like code smell but not really.

The problem of false negatives, where smelly code is wrongly seen as clean, is worse for tools that judge code quality. The Random Forest does not see 73 smelly examples out of 500 (14.6), and the Neural Network misses 95 out of 500 smelly code samples (19). These missed examples often have small code smells that the CodeBERT tool does not pick up on because it is a subtle smell that depends on the wider code or needs to be understood in context. It might be a smell that is only stylistic and not structural, or one that is spread over more than one place in the code so that not all parts of the smell can be seen in one small code fragment. The Neural Network's bigger miss rate makes it worse, because it misses more bad code.

These error patterns show both models have the same trouble with the same types of hard cases, so the problem is not one or the other classifier but the limitations of the embedding approach itself. If the Neural Network shows fewer false alarms but more misses compared to the Random Forest, it means it has learned a much more strict boundary, and it will do better in some uses but worse in others.

6.5.2 Potential Sources

There are a few reasons why there are some mistakes in this and they are each a big challenge for getting code smell detection right by machine. First, the data you put in has limits. That is because CodeBERT can only work with 512 tokens at a time, which means if your code file is longer than that it will only get part of the info, leaving out important parts that could help tell if the code is clean or smelly. Also, sometimes the info might not show the right meaning, like

what the code was meant to do or how distant pieces of code relate to each other.

Second, overlapping class is an important issue in code smell finders. The line between good and smelly code is thin and hard to judge, as many code samples exist in a gray area where even good developers might disagree. Also, the multi label problem is very common for real world smells, with one code sample often showing multiple smell types or a partial smell. The binary class task used in these experiments makes this problem easier to deal with, but does not show the full complexity of how code is judged.

Third, the data set limits how well the results can apply to other cases. The 5,000-sample subset, though needed for the work, may not show all types of code patterns found in real projects. The sampling method, while fair, creates a fake spread that may not match the real-world class imbalance where smelly code could be less frequent. Also, wrong tags in the original data set, where some human graders might have disagreed or made mistakes, cause some doubt that spreads to what the model predicts. These limits tell us that, while the results look good now, they need to be taken with care when used outside this study.

6.6 Discussion

6.6.1 Effectiveness of CodeBERT Embeddings

The test results show how well CodeBERT embeddings work for code smell detection. Both models get about 78 of their points from just the embedding vectors, without any hand-rolled features, coding domain knowledge, or feature engineering. This is a good achievement since most traditional code smell checks often need lots of feature engineering, such as finding code metrics, pattern matching, and rule-based checks. Both very different machine learning algorithms (tree ensemble and neural network) do equally well, showing that the CodeBERT embeddings carry enough text or code meaning and shape to be good for this task right away.

The high result with little prep work shows that CodeBERT's pre-training on big code bases has taught it what to look for and how to use it in the code smell task. This ability to transfer knowledge from different code gives a lot of value because it means you dont need a tech expert to pick the best features and the model can learn from the code bases it was trained on. The fact that both models give similar results shows that the embeddings hold strong signals that work across different learning algorithms. But the 78 accuracy limit means that there are natural limits to what static code embeddings can do, and some extra context or features may be needed for better scores.

6.6.2 Model Selection Considerations

Deciding to use Random Forest or Neural Network depends on what you need and can do. Both models work about the same but have strengths in different places. Pick Random Forest if you

need to explain the decisions, as it shows what data was important and can trace back to how the model made its call, which helps developers trust the model. The training time of just 2.1 seconds makes Random Forest great if you need to test ideas fast, train often with new data, or don't have much time or power to run things. Also, if you need to catch more smelly code (0.86 recall compared to 0.81), then Random Forest is the better choice as it will find more problematic code than false alarms.

Neural Networks are better if you want both classes to do well, as their precision and recall are similar. Their design is also better for future work, such as end-to-end fine tuning of the CodeBERT model or adding more input types. When working with much bigger data, neural networks can gain from more data. Also, when the pattern you want to learn is going to be very complex and may need features to be connected in many ways for a decision tree to do well, neural networks might be better.

6.6.3 Comparison with Related Work

Compare my work to related studies as seen in the table above. This shows my results fit into the larger field of code smell detection work. My CodeBERT-based solutions reach 78.0 to 78.2 accuracy, which is better than old style machine learning works that get 70 to 75 accuracy on the same tasks. For instance, Alomari and others got about 70 to 75 accuracy using old ML ideas on a larger set of data (107,000 samples), but their task was multi-smell detection a harder job than my simple yes or no match.

Rule based methods, which use fixed rules and pattern finding, often reach 60%–70% accuracy and do not easily apply to code outside the expected patterns of the rule creators. My embedding based system has better accuracy and can handle new code better, as the model learned representations that adapt to code it was not explicitly told about. But, it is hard to compare results because data, test rules, and what is being tested can vary too much between studies. My results on a balanced test set of 5,000 examples might not work as well on large unbalanced sets, and the binary task I use here is easier than the multi label task in the real world.

Study	**Approach**	**Acc.**	**Dataset**	**Notes**
This Work	CodeBERT + RF	78.2%	5,000	Binary
This Work	CodeBERT + NN	78.0%	5,000	Binary
Alomari et al. [1]	Traditional ML	$\sim$70–75%	107K	Multi-smell
Other Studies	Rule-based	60–70%	Various	Limited generalization

6.6.4 Limitations and Challenges

There are some limits that stop these results from being more broad or useful. The data set is small with only 5,000 samples, and real code is much more varied than this shows. The class balance is set to be 50–50, but in real code it will be less in the minority class. The study only looks at Java code, not other code types. The project turns multi-label problems into binary ones, but in real code, many samples can have many types of smell at once.

A model downside is the high cost to get CodeBERT embeddings, they have to be made for each code sample and that might be too much for very big code bases. The 512 token limit means I do not get the full code file if it is bigger, so I lose part of the context. No fine tuning of CodeBERT means I use the pre trained version, not a version that was fine tuned for this task which could work better. Few hyperparameter tuning makes it seem like the two models are not yet at their best, it might be better tuned.

Discussion limitations include only one train–test split, which gives just one performance estimate and may miss performance differences across data splits. No external testing on new datasets means I cant know how well the model works in other codebases or fields. Qualitative errors are not deeply analyzed here, so I cannot identify specific failure cases that might point to systematic errors or lessons for model improvement.

6.6.5 Implications for Practice

This has several key use cases for real software work. For automating code reviews, these models can help find parts of the code that are risky to catch most of the smelly code you have. The models are not perfect, but getting 78 out of 100 right with everything else, and making sure you find 97 out of 100 with anything smelly is very good. So if you can find the most risky parts of code with most code bases, you can let the rest go and let the human see only the worst parts. The models are very fast, taking less than 0.01 seconds to run so they can be used while you work.

For quality gates on the pipelines of continuous integration, these models can be added to tell you if a code smell is there before the code is joined. This can stop bad code designs from entering the code base. These models can be like a early warning system, not to replace human code review but to add to it. For training, the models can give instant advice to coders still learning the right way to do things, which makes a loop that teaches coders to see and stay away from code smells as they code.

When it comes to using it live, the time to get a guess is less than 10 milliseconds, so tools you click and IDE add-ons are easy. If you want to keep it up to date with how people write code, you can retrain it once in a while. Combining it with tools like IDEs, code review apps, or build jobs is simple since it takes input and gives back an answer, but the process of generating the embedding data might need a bit more work. The Random Forests being easy to explain might be a key point for some coders to be on board because they can see why code

was marked.

6.7 Statistical Significance

6.7.1 Performance Stability

Performance stability analysis shows that both models perform in a stable and reliable way. Both of them score about 78 accuracy but with high consistency meaning its not a coincidence or the way the data was divided thats making the models perform so well, but it is because the models actually learn from the CodeBERT embeddings. The accuracy in relation to precision and recall and F1 score are all close to 0.78 0.79, and this shows that the models are well-calibrated and do not perform in a way that favors either the precision or the recall at the expense of the other.

The performance balanced over both classes just adds to how strong these results are. Neither model is doing super well on one class and bad on the other that would point toward overfitting or bad generalization. The fact that these patterns hold across two very different machine learning algorithms is a good sign that the results are real signals in the data and not artifacts of one method or the other. This balance is good for real world use because it means the models will do well on different code snippets and projects.

6.7.2 Confidence Intervals

Statistical analysis of the confidence intervals provides important context for interpreting the observed performance differences. With a test set of 1,000 samples, the 95% confidence interval for accuracy is approximately $\pm 2.5\%$, meaning that the true accuracy of either model is likely to fall within a range of about 2.5 percentage points above or below the observed value. This confidence interval width reflects the inherent uncertainty in performance estimation due to finite test set size.

The observed differences between Random Forest and Neural Network models (ranging from 0.001 to 0.007, or 0.1 to 0.7 percentage points) are all well within the confidence interval margin of $\pm 2.5\%$. This means that the performance differences are not statistically significant and could easily be due to random variation in the test set composition rather than genuine algorithmic differences. This finding reinforces the conclusion that both models achieve essentially equivalent performance, and the choice between them should be based on non-performance factors such as interpretability, training time, or deployment constraints rather than marginal accuracy differences.

6.8 Summary

This chapter showed a full test of CodeBERT embeddings to find code smell. I used two different classifiers: Random Forest and Neural Nets. Both models were able to get about 78 accuracy on a balanced 5,000 sample subset of the SmellyCode dataset. This proves that embedding work is a good way to find code smell without having to create hand-crafted features or know a lot about the domain. The Random Forest is slightly better for accuracy overall (78.2 vs 78.0), but these results are so close that it does not matter much and they are well within what you expect.

The detailed analysis showed trade offs of the two options. The Random Forest is better if recall for smelly code is more important (0.86). It is better if you want to make it faster and easier to understand (2.1 seconds to train). The RF is good if you want to test things fast and have a model that is clear and easy to read. It is good if you need a model that is simple but can work for many things.

The Neural Net is better if you want to try to do better on both classes (more balanced precision and recall). It takes longer to train (5–10 minutes) but it may do better in the future if the model can be changed to do better or if you want to try it on larger sets.

The embedding-based method shows that it can be used in the field. The time it takes to make a decision is less than 0.01 seconds, so it could work in the moment. The results are good compared to machine learning methods (70-75% accuracy) and rule based methods (60-70% accuracy). It also works better on new code. Still there are limits that stop it from being used on everything.

The data set it was tested on is small and made so that each group of code has the same number of samples. It only works on Java code. The simple yes-or-no answer makes it hard to know how accurate it is. The work shows there is good value in using CodeBERT's new features for identifying code smells. The results prove that the new features can find useful information about code.

The results show there is a lot of potential in using CodeBERT for detecting code smells.

7 Conclusion and Future Work

This part is a full overview of my work in this study, where I summed up my findings from my research, including the data, how I did it, and what I found out about using CodeBERT embeddings to find code smell. I gone through each question I had and look at the limits of my study and give clear ideas on how to move this research forward. I shifted from what I did right now to how it can help software engineers and future research.

7.1 Conclusion

Our study has shown that combining CodeBERT's 768 sized embeddings with regular machine learning tools and neural networks is a good and useful way to do code smell detection of Java code. The key result comes from my tests and trials: the fixed 768 sized embeddings, taken from prebuilt CodeBERT models and used without any hand held features or guidance, give enough good meaning to tell if code smells are present in binary form. Both Random Forest and a simple shallow MLP showed good scores for accuracy while being fast enough to work in real time. This mix of good scores and speed means that the method is not just a research idea but a real choice for use in CI jobs or in developer tools.

But this finding matters more than just for smelling. It shows that it's possible to learn from very large sets of code and find the things that matter for software quality. It also helps show that you don't need really fancy models to tell you things about the code. Seeing that both tree-based and neural methods did about the same is proof that the embedding space has useful info in it, you don't need to build very complex models to find what you need. This is good for when you want to run code quality checks without much resources, simpler models are easier to read and faster.

7.1.1 Summary of Contributions

Our work adds to the field of software engineering research in many ways both in technical and scholarly form and content.The contributions come from many places from new uses of known ideas to ideas that matter for engineering applications.

Technical Contributions. My main technical point is that I showed that feature extraction from CodeBERT is a good way to find code smell.Our method does not use old measures like cyclomatic complexity, coupling, or cohesion instead it uses the code meanings that the transformer model was trained on CodeBERT was trained on millions of code-comment pairs in many different languages it learns many of the relationships between what code looks like,

what it means, and what it is supposed to do I found that these 768 part vectors hold enough information to tell if a program is smelly or not, and that the model learned some kind of codequality pattern when it was being trained.

Another point on technique is the side by side test I did on modeling. I tested and compared the Random Forest and a simple 3 layer neural net classifier and give real evidence on why one might choose one over the other in a deployment setting. Random Forest has the advantage of explainability and robustness to hyperparameter choices, while the MLP may give a better approximation of the data that can better fit the non-linear structure of the embedding space. The fact that the two models perform in a similar way suggests that one can choose the right classifier based on deployment constraints and not on performance.

Last I built an end to end pipeline that covers data prep, embedding extraction, training, and testing. That pipeline is designed to be repeatable with clear notes on data processing, hyperparameters, and testing procedures. The outputs of that pipeline, including trained models, test scores, and inference code, are laid out in a way that should make it easy to add to any app or QA process. It is built modularly so that researchers and engineers can swap out any one piece while keeping things compatible with the rest.

Methodological Contributions. From a scientific view, my work on balanced samples in code smell datasets is important. In the real world, code bases are very uneven, with clean code often outnumbering smelly code a lot. This makes it easy for models to get high accuracy just by guessing the simple answer, but they won't pick up the actual bug-prone code devs want to catch. I handled this with stratified subset creation, making sure I pick examples carefully so the training data is fair and can teach models both parts of the problem, though the 50–50 split isn't true to life in a work setting.

In my tests I used many different scores that show different parts of how the model works. If I only use accuracy it might not show all the info I need because the test can be unbalanced, so I also give Precision, Recall, and F1 scores for both classes. When I use more than one score it shows more info that accuracy would hide. For example, the test can be very accurate but if it misses many smelly code it will show a very low recall score for the positive class. When I look at the score for each class I see more info about where the model might mess up and what I can do to fix that. With the confusion matrices and class scores I can learn more and see where the models might make the most errors.

7.1.2 Key Findings

My tests shows many things I learned about how to find code smell with learned words, and my first point is about how well the words work for this job. I've shown that 768 length CodeBERT words have enough info to say if there is a smell or not without having to do much work on top of the word extraction. This is great news because it shows that the words can tell us about

code smells and that CodeBERT was not made for this kind of work but it still can do it well. It looks like the words have info about how the code is built, how hard it is, and how easy it is to change, and this info lines up well with what pros think about code smells. This is useful because it makes it easier to set up smell detectors and do this kind of work without needing to spend a lot of time fine tuning different measurements.

A second finding is about model parity for rf and mlp classifiers. while their architectures and learning processes are very different, the result is very similar: both models do well in my test set and differences are not statistically significant. this means that the embedding space itself is a strong signal to classify the data, and that both models can learn a good decision boundary. in my experiments, rf did show a small numeric edge, but that difference is within the randomness expected in initial weights or data sampling. this parity has value: users can pick models based on how they will deploy the work, need for transparent rules, or computation needs, instead of being forced to pick an architecture for fear of accuracy.

The third point is about the **complementary strengths** of both method. random forest had better recall for the smelly code class, it was better at finding the broken code parts, though it also misclassified some good code parts. this makes it good for quality checking, where missing a smell is more costly than flagging a healthy code part by mistake. Random Forest also can show the most important features, the parts of the code writing that matter most to the class decision. The MLP was more balanced, both classes had similar recall and precision, it was more fair but less sensitive. this balanced approach could be better when missing something is just as bad as flagging something wrong by mistake.

Lastly, I found that this method is easy to use in real life. The time it takes for the models to give an answer shows they can be used for code blocks in a matter of milliseconds, making it good for tools that check code right as a programmer writes it. The models dont need a lot of computing power, so they can run on most regular computers used by programmers today. It is also easy to add the models into systems that automatically check code before it is added to a project, because it does not slow down the process too much. The way the models are set up is simple, needing only a pre-made CodeBERT model and a trained classifier, which is easier than more complex setups that might need several models or complex steps.

7.1.3 Limitations

Our work shows good results but there are some limits you should know so I give you a full, honest review of the scope of this research and how far it can go.

Dataset Limitations. The biggest problem is about my **dataset composition**. I did my tests on a set of about 5000 samples from a lot larger collection of Java code. This is enough for a first look but it is only part of the data, and if you go to the full collection the results might be different or you might need a different kind of model. Even more serious is I made a plain

balanced data set with equal parts smelly and clean code which does not match the real world count of clean to smelly code in open source repos. Usually, clean code is most of what you see, and a model trained on balanced data may not work well in the real world. A next step for this work should be to test how performance drops if the model is trained on balanced data and then used on a natural imbalanced distribution, and see if techniques like cost sensitive training or changing the decision threshold can help.

I have some limits in my data. My data is limited to Java code, so my finders might not work so well for other languages. The pattern for code smells might change from one programming language to another because of language rules, common use, and best practices. My labeling system is very broad, only telling if code has any smell at all or if it is clean. This makes it hard for the system to tell you which refactoring may help, which makes the tool less useful for a programmer trying to fix code.

Model Limitations. Our model have some limits I could fix in the future. I picked CodeBERT as a frozen encoder this means I did not train the transformer weights for the smell task. This is fast to train and I get to use the knowledge from the pretraining, but it could do better if I fine tune on the task like the code for the smell. The size was kept to 512 tokens, meaning longer code could get cut off and I could lose info from classes or functions longer than 512 words. This is a limit when I want to find smells like Long Method or God Class where the size is part of the smell.

I also shown that I only looked a little bit at changing the settings. While I tried some settings to find good ones, I didnt do a full grid search or use bayesian optimization to find best ones. The models I showed are good ones that I found by trying different ones but they might not be the best ones. Also I didnt try combining predictions from more than one model which might help make better predictions because the errors of the models might be different in different places.

Evaluation Limitations. My method of evaluation, while wide in how many numbers I look at, can be counted short in how I checked it. I used a single train and test split, which shows us a snapshot of how well the model performs but doesn't show us how different splits would affect my numbers. Using cross validation would give better signals on how well my model does and how steady my results are. Not having a second set of data from a different place or time to test on makes it hard to know how well it would work on new code or on new types of code in the future.

Also, my qualitative error checks were only on general confusion tables and scoring data. A more detailed look at actual wrong guesses would tell us more about what kinds of code give the models trouble, which might show us a pattern of bias or failure. Why do some parts of code get misjudged will give clues for fixing the embedding process, design, or training.

7.2 Research Questions Answered

Our study was based on three basic questions that I now answer step by step on what my experiments showed us.

RQ1: Can CodeBERT embeddings detect code smells effectively? Our test results show this question has a clear yes answer. These 768 fact vectors, when fed into common classifiers, give good baseline accuracy for binary code smell detection. The main idea is that the info inside these 768 figures carries enough clues to separate smelly code from clean code, even if no one has taken the time to craft special features or metrics for the problem. This is because CodeBERT had to learn to understand code before it knew how to do anything with it, and in doing so, it built representations that show how code looks structurally, what names mean what, and how code blocks fit together that have a link to how good or bad the code is.

The numbers show that this is true in my tests, where both the Random Forest and the MLP had good scores that are as good or better than methods based on carefully selected features. Its great that the embeddings alone can do this well without extra feature work, and this proves that learning from code tasks can also help in quality tests. This is important for real world use, as it means less work to make the smell detection work, and that it can be used by teams that are not experts in software metrics or feature work.

RQ2: Which classifier is better (RF vs. MLP)? Our analysis shows that Random Forest and MLP classifiers are pretty close in performance for code smell detection, with Random Forest having a slight edge in numeric value but within the expected margin of error if data was random. This similarity implies the embedding space holds good info for this task and that both models are effectively learning meaningful boundaries. So, which to pick shouldn't rely solely on accuracy, but consider factors like explanation needs, use cases, or processing ability.

Random Forest is better for reading how the model works because it shows what parts of the code embeddings it uses to find the smell. It is also easier to get to work well because it does not need as much changing of the settings. The MLP is harder to read, but it might find more complex non-linear patterns in the big sets of code data. In my tests, the MLP had more even results across classes, which could be better if a person does not want to have a lot of false alarms but does not want to miss many true bugs either.

RQ3: Is this approach practical? Our method shows it is practical in many ways. Accuracy tests show both models can analyze code in just milliseconds, which helps when providing instant feedback in tools like IDEs. The computational needs are low, so most computers used by programmers can run my models without problem. The design is simple, needing only a pre-trained CodeBERT and a trained classifier, which means less setup time and ongoing work compared to more complex options.

The real benefits go beyond just the technical feasibility they include how easy it is to add to your tools and keep it up to date. My pipeline is built in a way that each part can be updated on its own, and it uses common machine learning tools that most people are familiar with. The fact that my approach needs no hand-made features means people won't need special knowledge about the software metrics, making it easier for more teams and firms to use. All of these points show that my method isn't just something to study in class, but is really useful if you want to put it in the day to day work of making software and pushing code changes into the main branch.

7.3 Future Work

Our work today is good, but I can add many new things to make it better. I put these new things in four groups: dataset and tests, model things, tech things, and big research things that make more use for it all.

7.3.1 Dataset and Evaluation

The next step is scaling all the way to fullcorpus training my experiments so far have used only part of the data I have, because it took too long to embed all the code, and to batch it all up in one go or to run the whole thing on a GPU. But if I trained the full corpus, I might see a pattern that is only right there and not in the smaller data set, and that might give us a better generaliser because it can see more different styles and sorts of code. It might also let us make more complex classifiers that might overfit on small data.

Another good way to go is multi label detection of certain smell types. My binary classification system, while good for a initial check, loses the detail needed for more specific feedback. If I add the ability to guess specific smells like Long Method, God Class, Feature Envy, or Data Clumps it would help developers know more about what refactors they can make. To do this it would need multi label classification tools and maybe hierarchical classification methods that can see links between different smell types. The problem is code pieces can smell like more than one thing at the same time, so models would need to handle overlapping labels and maybe even signals that don't agree.

To make my results more reliable and more true, future work should include better tests. K-fold cross-validation would give more steady estimates of how well the model does and give a better picture of how the model might vary across different sets of data. External testing with data from other places, times, or coding backgrounds would show how well the model works on new code or when coding styles change. Perhaps most boldly, language transfer tests could see if models trained on Java code can find smells in Python, C++, or other languages this might be possible because CodeBERT is already trained on more than one language and can be used to make tools for finding quality issues in different languages.

7.3.2 Modeling Enhancements

The best idea for improving the model is to tune CodeBERT just for the code smell task. My method now uses CodeBERT as a frozen feature extractor, which uses what the model learned about code when it was pretrained, but it does not change the embeddings to match smells. To fine-tune would mean training the transformer's weights from start to finish with the classifier on top, so the model can learn task-specific ideas that might be better for the delicate smells. I would have to do it carefully so the model does not forget the good code knowledge, but it could do much better.

Other design choices in the model could also be better. For example, heads that use attention are easier to explain because they can show which parts of the input code or which words in the input text matter most. Classifiers based on transformer models that look at sequences of word or code representations rather than single fixed representations might show how different parts of a code piece relate to each other. For code bases that have versions over time, time-based models could follow how smells change across different code versions, even catching signs of bad smells before they become worse. But such design choices would need to be checked to see if they make the model too slow or complex.

A third option is to mix the learned features with other traditional methods of software scoring and AST features. My work shows the features used alone work well, but mixing two or more signals might be better and more reliable, and could give new insight. Software score metrics like cyclomatic complexity, coupling, or cohesion look at how the code is built in a way that might not be fully covered by the semantic features alone. AST features can give clear structural details that might match the learned features. The problem is how to bring together these different types of features, which could need special designs for the overall network or learning process.

7.3.3 Technical Improvements

There are some tech things that would make the system better and easier to use. The 512 token sequence length limit is a big problem when looking for smells in longer code. This limit could be fixed by chunking longer methods or classes into overlapping parts, using hierarchical encoders that describe code in different levels of detail, or sliding window approaches that keep the context across chunks. Each of these ways would have different costs and challenges and would need to be tested in practice.

Productionization efforts would turn the research prototype into a real system. This would include building REST inference services that can be plugged into existing toolchains, creating IDE plugins that show feedback in the editor while developers are writing code, and setting up CI quality gates to automatically flag bad code before it gets merged into main. For models going into production, monitoring infrastructure is also needed to track how well the models perform over time, alert when data drifts as coding styles change, and identify when

retraining might be worth doing. These engineering requirements are part of turning the proof of concept into usable tooling.

Explainability is a key thing developer would want to see. Even if the model is very good at correctness, developer wants to know why it is saying there is a smell so they know what action to take. They could use SHAP or LIME techniques after prediction to explain why the model thinks a piece of code is smelly. They could also use attention probing to see where the model is looking when it decides. If the system could highlight the exact lines or parts of code that make it think a smell is present, it would go from being a black box to being an easy tool that developer can use to get useful feedback.

7.3.4 Research Extensions and Applications

There are other research ideas that can make this work bigger or more useful, not just the technical stuff. Severity prediction would list detected smells based on how likely they are to affect maintainability, testability, or other quality factors. Developers would then know which smells need to be refactored first or where to focus their efforts on the high-severity smells that pose the most risks. Prediction of severity might include many different factors, like how often the code is run or how complex the code around the smell is or what the history is of similar smells in the code base.

Evolution tracking could allow long term view of smells as code evolves so you can see how code gets worse or better over time. This helps you find early signs your code might be headed towards maintenability problems so you can fix the code before it gets that bad. Evolution tracking might also show links between how people code, who works on the project, or what part of the project they are on that leads to smells in code. This can tell you more about how code quality forms, and what parts of projects and how they are coded affect quality.

Developer education is one way it can be used. Its not just to find but also to learn and get better. When developers get feedback in real time it can help them learn why these smells matter and how they can do something else. An educational system can teach developers about how code should be good and over time they can get better at knowing why this smells and why it does not. This way the tool can work as a quality gate and a learning platform that can make developers better at writing code.

In conclusion, if I make a open site with standards and open source objects it will be more easy to reproduce more research and the comunity will be able to work on this project. Standardized testing rules would make it easier to compare different approaches, and open source implementations would make it easier to use and improve. This site could have organized data, testing systems, and starting solutions for future works that could help the growth in automatic code quality grading.

7.4 Summary

This work shows that CodeBERT embeddings with small classifiers can give a strong useful guide line for automated detection of code smell. The method makes transfer learning from big code understanding tasks work for quality work, and it saves people the trouble of making features by hand and still gets good results. Both RF and MLP show how well they work, and they work in different ways, so users can pick which one they want based on how they will use it, not on accuracy issues.

The approach is practical because it works right now with real-time inference and is easy to set up and use, which makes it a real tool for the coding world and CI pipelines. But this is just the start, there is much more room to make it better with more data, fine-tuning, handling longer code better, and making it more clear.

As automated code quality tools get better, this work sets the base for many more steps. If given more data, tools fine-tune for a specific task, handle longer code lines, and can explain what they think, then this method could grow into a full quality signal that gives coders ideas that are easy to understand about how to make code easier to work with in the long run. Including these tools in workflows and CI systems is a good step toward making code better and less costly over time, and to help keep code in good shape for the future.

Bibliography

[1] N. Alomari, A. Alazba, H. Aljamaan, and M. Alshayeb, "SmellyCode++.csv," figshare dataset, 2025.

[2] Z. Feng et al., "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," *Findings of EMNLP*, 2020.

[3] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[4] M. V. Mäntylä, J. Vanhanen, and C. Lassenius, "Bad smells in software—a taxonomy and an empirical study," in *CSMR*, 2004, pp. 119–128.

[5] F. Palomba et al., "Detecting bad smells in source code using change history information," in *ASE*, 2013, pp. 268–278.

[6] A. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?" in *ICSM*, 2012, pp. 306–315.

[7] F. Zhang et al., "Towards building a universal defect prediction model," in *MSR*, 2014, pp. 182–191.

[8] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *ICSM*, 2004, pp. 350–359.

[9] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*. Springer, 2006.

[10] M. H. Halstead, *Elements of Software Science*. Elsevier, 1977.

[11] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. 2, no. 4, pp. 308–320, 1976.

[12] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, 1994.

[13] F. Khomh et al., "An exploratory study of the impact of code smells on software change-proneness," in *WCRE*, 2009, pp. 75–84.

[14] SonarSource SA, "SonarQube: Code Quality and Security," 2024. [Online]. Available: sonarqube.org

[15] M. Tufano et al., "When and why your code starts to smell bad," in *ICSE*, 2015, pp. 403–414.

[16] G. Rasool and Z. Arshad, "A review of code smell mining techniques," *Journal of Software: Evolution and Process*, vol. 29, no. 11, 2017.

[17] M. Lanza, R. Marinescu, and S. Ducasse, *Object-Oriented Metrics in Practice*. Springer, 2006.

[18] F. A. Fontana and M. Zanoni, "Code smell severity classification using machine learning techniques," *Knowl.-Based Syst.*, vol. 128, pp. 43–58, 2017.

[19] D. Di Nucci et al., "A developer centered bug prediction model," *IEEE Trans. Softw. Eng.*, 44(5), 2018.

[20] F. A. Fontana, P. Braione, and M. Zanoni, "Automatic detection of bad smells in code: An experimental assessment," *J. Object Technol.*, vol. 11, no. 2, 2012.

[21] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empir. Softw. Eng.*, vol. 21, no. 3, 2016.

[22] A. Maiga et al., "SMURF: A SVM-based incremental anti-pattern detection approach," in *WCRE*, 2012, pp. 466–475.

[23] N. Khleel and K. Nehéz, "Detection of Code Smells Using Machine Learning Techniques Combined with Data-Balancing Methods," *Int. J. Adv. Intell. Paradigms*, 2024.

[24] M. A. R. Bhuiyan et al., "Code Smell Detection via Pearson Correlation and ML Hyperparameter Optimization," arXiv:2510.05835, 2025.

[25] A. Alazba et al., "An Empirical Evaluation of Ensemble Models for Python Code Smell Detection," *Applied Sciences*, 15(13), 7472, 2024.

[26] L. Breiman, "Random forests," *Mach. Learn.*, 45(1), 5–32, 2001.

[27] T. G. Dietterich, "Ensemble methods in machine learning," in *MCS*, 2000.

[28] A. Gupta et al., "Unsupervised Machine Learning for Effective Code Smell Detection," *J. Computer Science*, 20(12), 2024.

[29] M. Allamanis et al., "A survey of machine learning for big code and naturalness," *ACM Comput. Surv.*, 51(4), 2018.

[30] A. Hindle et al., "On the naturalness of software," *Commun. ACM*, 59(5), 2016.

[31] A. Agrawal et al., "Deep Learning for Source Code Modeling and Smell Detection," arXiv:2009.09176, 2020.

[32] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, 521, 436–444, 2015.

[33] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *ICLR*, 2018.

[34] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, 9(8), 1735–1780, 1997.

[35] J. Li et al., "Code completion with neural attention and pointer networks," in *IJCAI*, 2018.

[36] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *ICLR*, 2015.

[37] H. Husain et al., "CodeSearchNet challenge: Evaluating the state of semantic code search," arXiv:1909.09436, 2019.

[38] M. Tufano et al., "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *TOSEM*, 28(4), 2019.

[39] D. Guo et al., "GraphCodeBERT: Pre-training Code Representations with Data Flow," in *ICLR*, 2021.

[40] A. Kanade et al., "Learning and evaluating contextual embedding of source code," in *ICML*, 2020.

[41] W. U. Ahmad et al., "Unified pre-training for program understanding and generation," in *NAACL*, 2021.

[42] E. Nijkamp et al., "CodeGen: An open large language model for code with multi-turn program synthesis," in *ICLR*, 2023.

[43] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE TKDE*, 21(9), 2009.

[44] A. Nandani, A. Sharma, and A. Chhabra, "DACOS: A Dataset for Code Smell Detection," arXiv:2303.08729, 2023.

[45] R. Tufano et al., "An empirical investigation into the nature of test smells," in *ASE*, 2016.

[46] G. Rasool and P. Mader, "A customizable approach to assess code smell severity," in *SANER*, 2015.

[47] N. V. Chawla et al., "SMOTE: synthetic minority over-sampling technique," *JAIR*, 16, 321–357, 2002.

[48] G. E. Batista, R. C. Prati, and M. C. Monard, "A study of the behavior of several methods for balancing machine learning training data," *SIGKDD Explor.*, 6(1), 20–29, 2004.

[49] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*. Springer, 2006.

[50] M. Tufano et al., "When and why your code starts to smell bad (and whether the smells predict faults)," *IEEE TSE*, 43(11), 2017.

[51] B. Kitchenham and S. L. Pfleeger, "Personal opinion surveys," in *Guide to Advanced Empirical Software Engineering*, Springer, 2008.

[52] T. Saito and M. Rehmsmeier, "The precision-recall plot is more informative than the ROC plot," *PLoS ONE*, 10(3), 2015.

[53] S. Herbold, A. Trautsch, and J. Grabowski, "Global vs. local models for cross-project defect prediction," *Empir. Softw. Eng.*, 22(4), 2017.

[54] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *ICML*, 2014.

[55] J. Devlin et al., "BERT: Pre-training of deep bidirectional transformers," in *NAACL*, 2019.

[56] A. Vaswani et al., "Attention is all you need," in *NeurIPS*, 2017.

[57] M. I. Azeem et al., "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis," *Inf. Softw. Technol.*, 108, 2019.

[58] S. Yadav, S. Kumar, and A. Chhabra, "Machine Learning-Based Methods for Code Smell Detection: A Systematic Review," *Applied Sciences*, 14(14), 6149, 2024.

[59] M. Tufano et al., "An empirical investigation into the nature of test smells," *TOSEM*, 28(4), 2019.

[60] F. Palomba et al., "Landfill: An open dataset of code smells with public evaluation," in *MSR*, 2015.

[61] L. Breiman, "Random forests," *Mach. Learn.*, 45(1), 2001.

[62] H. Peng et al., "Building program vector representations for deep learning," in *KSEM*, 2015.

[63] M. Allamanis et al., "Suggesting accurate method and class names," in *FSE*, 2015.

[64] N. Alomari, A. Alazba, H. Aljamaan, and M. Alshayeb, "SmellyCode++: A large-scale dataset for code smell detection," in *figshare*, 2025.

[65] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of EMNLP*, 2020.

[66] M. Fowler, K. Beck, J. Brant, and W. Opdyke, “Refactoring: Improving the design of existing code,” *Addison–Wesley*, 1999.

[67] A. Author and B. Author, “Limitations of rule-based static analysis for code quality,” in *Proc. of Conference/Journal*, Year.

[68] C. Author, D. Author, and E. Author, “On the impact of code smells on defects and maintenance effort,” in *Proc. of Conference/Journal*, Year.

[69] M. Allamanis et al., “Suggesting accurate method and class names,” in *FSE*, 2015.