# Severity Classification of Software Code Smells Using Machine Learning Techniques

**Prof. Gaurav Singal, Sanya Garg, Kairavi, Anurag Agarwal**
*Netaji Subhas University of Technology*

---

**Abstract**

Code smells are signs of poor design choices in software that can hurt its overall quality. It's important to measure the severity of these smells so developers can prioritize which parts of the code need refactoring. While there has been a lot of research on detecting errors in design patterns, there hasn't been much focus on how to measure the severity of code smells or which machine learning models work best for this. This paper aims to address that gap by using different machine learning techniques, including regression, multinomial, and ordinal classification models, to classify the severity of code smells. We also use the Local Interpretable Model-Agnostic Explanations (LIME) algorithm to make the predictions of the models easier to understand.

**Keywords:**
Code smell, LIME, sampling, regression models, severity classification, software refactoring, multinomial, ordinal, random forest, naïve Bayes,XGBoost, SMOTE

---

## 1. Introduction

### 1.1 Introduction to Domain

Code smell is a term used to describe structural characteristics in software that indicate potential problems in the code or design, which might complicate software maintenance. Unlike bugs, code smells do not necessarily prevent the program from running, but they can make the code more prone to failure or performance issues. In large-scale software projects, detecting and prioritizing code smells during the development phase is essential, as it helps in managing software maintenance more effectively. Early detection of code smell severity aids developers in focusing on which issues should be addressed first and optimizes refactoring efforts for better software quality.

### 1.2 Problem Description

Despite the significant advancements in code smell detection techniques, several limitations persist that hinder their widespread applicability and effectiveness. One of the primary challenges is the inherent subjectivity in defining what constitutes a "code smell." Different developers, teams, and tools may have varying interpretations of code smells, which leads to inconsistent detection results. Furthermore, the criteria and thresholds applied by different detectors are not standardized, which can result in false positives or missed detections. Many existing detectors also rely on a set of predefined rules that are not adaptable to all types of projects or domains, further complicating the detection process. Additionally, while some detectors consider basic code metrics, they often overlook critical system design details, such as architectural patterns and the interrelationships between different components, which can significantly influence code quality. This lack of context awareness can lead to discrepancies in the results, making it difficult for developers to accurately assess the severity of the detected smells. As a result, there is a pressing need for more precise, consistent, and automated methods for detecting and classifying the severity of code smells, which can better integrate with real-world development practices and provide more actionable insights for refactoring and improving software quality.

### 1.3 Motivation/Objective

The motivation behind studying code smell severity classification is multifaceted, with significant implications for both academic research and practical applications in the software industry. For computer science students and researchers, this study provides a valuable learning tool for understanding the concept of code smells, their impact on software quality, and methods for addressing them through refactoring. In the software development industry, professionals such as developers, architects, and technical leads are tasked with maintaining codebases that are not only functional but also maintainable and scalable. By classifying the severity of code smells, this study offers insights into improving code readability and reducing technical debt. Furthermore, open-source contributors and large-scale software development communities can leverage this research to enhance collaboration and ensure that their code adheres to best practices, ultimately improving the overall quality of shared software. Quality assurance teams can utilize these findings to identify areas for refactoring and better assess the quality of the code. Finally, this research opens avenues for further exploration within the research community, contributing to the development of more effective methodologies for detecting and addressing code smells. By advancing the understanding of code smell severity, this study aims to provide a foundation for improving software quality across various domains.

### 1.4 Contribution

This study makes several contributions to the field of software engineering, particularly in the area of code smell detection and severity classification. The key contributions are as follows:

- Development of Machine Learning Models for Code Smell Severity Classification: This research presents the development and implementation of machine learning models tailored to classify the severity of four prevalent code smells—God Class, Data Class, Feature Envy, and Long Method. By accurately predicting the severity of these code smells, this study provides developers with a tool to identify areas in their codebase that require refactoring and attention.

- Comparison of Different Machine Learning Approaches: The study compares the performance of multiple machine learning techniques, including multinomial, ordinal, and regression models, to classify code smell severity. This comparative analysis helps identify which approach is most effective for predicting the severity of different code smells, contributing to the body of knowledge on how various algorithms handle software quality issues.

- Improvement of Model Accuracy Using SMOTE Resampling: To address the issue of class imbalance in the dataset, the study applies the SMOTE (Synthetic Minority Over-sampling Technique) resampling method. By balancing the dataset, the study enhances the accuracy and robustness of the machine learning models, providing more reliable predictions for code smell severity classification.

- Interpretability through LIME Algorithm: To promote transparency in machine learning decision-making, this research employs the LIME (Local Interpretable Model-agnostic Explanations) algorithm. LIME helps interpret the predictions of machine learning models, enabling developers to better understand why certain code smells are classified with specific severity levels. This interpretability is crucial for practitioners who need to trust and act on the model's recommendations.

### 1.5 Paper organization

The remainder of this paper is structured as follows:

- Section 2: Literature/Related Work : This section provides a comprehensive review of existing research related to code smell detection and severity classification. It highlights the methods, techniques, and machine learning approaches that have been explored in the literature for identifying and addressing code smells. This review also discusses the gaps and limitations in current approaches, setting the stage for the contributions made in this study.

- Section 3: Dataset Description : Here, we describe the dataset used for the study, including the types of code smells represented, the metrics collected, and the data preprocessing steps taken to prepare it for

analysis.

- Section 4: Methodology : In this section, we present the methodology adopted for the study. It outlines the various machine learning models (multinomial, ordinal, and regression) that are applied for classifying code smell severity. We detail the data preprocessing steps, feature selection, model training, and the techniques used to enhance model performance, including SMOTE resampling and the LIME algorithm for interpretability. Additionally, we describe the experimental design, including the dataset used, feature engineering strategies, and the classification criteria.

- Section 5: Result Analysis This section discusses the results obtained from the experiments conducted with the machine learning models. We provide detailed comparisons of the performance of the different models across various evaluation metrics (e.g., accuracy, Spearman's correlation, RMSE, and MAE). Graphical representations of the results are provided to visually compare the model performances. The section concludes with an analysis of the results, including insights into the effectiveness of the different approaches and techniques.

- Section 6: Conclusion and Future Work In this section, we summarize the key findings of the study and provide conclusions drawn from the experimental results. We also discuss the limitations of the current study and suggest potential avenues for future research, such as improvements in model accuracy, better feature engineering, and exploring other machine learning techniques for code smell severity classification.

Section 7: References This section lists all the references cited throughout the paper, including research papers, books, and other academic sources that contributed to the background, methodology, and results of this study.

**2. Literature/Related Work**

## 0.1 Paper 1: Bad Smell Detection Using Machine Learning Techniques

`https://link.springer.com/article/10.1007/s10664-023-10436-2`

**Problem:** The study addresses the challenge of detecting code smells—indicators of design issues that affect software maintainability, scalability, and understandability—traditionally identified through inefficient manual processes. The role of automated solutions in improving this detection process remains underexplored.

**How:**The authors systematically reviewed 17 studies to assess the effectiveness of various machine learning (ML) algorithms in enhancing code smell detection. They trained models on datasets of known code smells, examining algorithms such as Support Vector Machines (SVM), J48 decision trees, and Random Forests using tools like WEKA.

**Why:** The goal was to develop reliable, scalable tools for automatic code smell detection, facilitating early refactoring and ultimately improving software quality. Machine learning techniques were investigated for their potential to increase accuracy and reduce human bias in the detection process.

**Idea:** The key idea is that machine learning can automate the detection of code smells, thereby increasing detection accuracy and speed while reducing reliance on inefficient manual processes.

**Advantages:** The study found that machine learning, particularly with algorithms like J48 and Random Forests, significantly increases accuracy and speed in detecting code smells. It can handle large datasets effectively and reduces human bias in the detection process.

**Disadvantages:** Challenges include inconsistent datasets and limited scalability, as well as varying accuracy for different types of code smells. The exploration of ensemble learning methods did not consistently improve performance

## 0.2 Paper 2 : Comparing and experimenting machine learning techniques for code smell detection

`https://link.springer.com/article/10.1007/s10664-015-9378-4`

**Problem:** The study addresses the challenge of detecting code smells—indicators of design issues that affect software maintainability, scalability, and understandability—traditionally identified through inefficient manual processes. The role of automated solutions in improving this detection process remains underexplored.

**How:** The authors systematically reviewed 17 studies to assess the effectiveness of various machine learning (ML) algorithms in enhancing code smell detection. They trained models on datasets of known code smells, examining algorithms such as Support Vector Machines (SVM), J48 decision trees, and Random Forests using tools like WEKA.

**Why:** The goal was to develop reliable, scalable tools for automatic code smell detection, facilitating early refactoring and ultimately improving software quality. Machine learning techniques were investigated for their potential to increase accuracy and reduce human bias in the detection process.

**Idea:** The key idea is that machine learning can automate the detection of code smells, thereby increasing detection accuracy and speed while reducing reliance on inefficient manual processes.

**Advantages:** The study found that machine learning, particularly with algorithms like J48 and Random Forests, significantly increases accuracy and speed in detecting code smells. It can handle large datasets effectively and reduces human bias in the detection process.

**Disadvantages:** Challenges include inconsistent datasets and limited scalability, as well as varying accuracy for different types of code smells. The exploration of ensemble learning methods did not consistently improve performance.

## 0.3 Paper 3: Improving accuracy of code smells detection using machine learning with data balancing techniques

`https://link.springer.com/article/10.1007/s11227-024-06265-9`

**Problem:** The paper addresses the challenge of detecting code smells—suboptimal software design elements—using machine learning (ML) and deep learning (DL) models, which often struggle with class imbalance in datasets where instances of code smells are significantly fewer than those of non-code smells.

**How:** To tackle this issue, the authors propose combining two deep learning models, Bidirectional Long Short-Term Memory (Bi-LSTM) and Gated Recurrent Unit (GRU), with data balancing techniques such as random oversampling and Tomek links.

**Why:** The goal is to improve the detection of four specific code smells (God Class, Data Class, Feature Envy, and Long Method) by mitigating the class imbalance, ultimately leading to better performance in the

detection process.

**Idea:** The key idea is to enhance code smell detection accuracy by integrating deep learning models with data balancing techniques to handle class imbalance effectively.

**Advantages:** This approach leads to significantly better performance, achieving up to 100% accuracy on balanced datasets for some code smells, along with higher precision, recall, and F-measure scores.

**Disadvantages:** Potential drawbacks include the introduction of bias through oversampling, the risk of losing data via undersampling, and increased computational complexity associated with the proposed methods.

## 0.4 Paper 4: A severity assessment of python code smells

`https://ieeexplore.ieee.org/abstract/document/10295478`

**Problem:** The research paper examines poor coding practices, known as code smells, in Python software, which adversely affect software quality, maintainability, and functionality.

**How:** The study analyzes five specific code smells—Cognitive Complexity, Collapsible "if" Statements, Many Parameters List, Naming Conventions, and Unused Variables—across 20 open-source Python projects with over 10,550 classes. It employs machine learning, specifically the J48 algorithm with AdaBoost, to classify the severity of these code smells, achieving an accuracy of 92.98%.

**Why:** The goal is to assess the severity of these code smells across different software development phases to highlight their impact on software quality and the necessity of managing them effectively to prevent degradation.

**Idea:** The key idea is to provide a data-driven approach for developers to prioritize refactoring efforts based on the severity of identified code smells, thereby improving overall software quality.

**Advantages:** The study offers valuable insights into the evolution of code smells and demonstrates that some smells, like Cognitive Complexity and Many Parameters List, became less severe over time, while others worsened. This information can help developers focus their refactoring efforts more strategically.

**Disadvantages:** The focus on only five code smells and reliance on SonarQube for analysis may limit the broader applicability of the findings. Additionally, the specific contexts of the analyzed projects may not generalize to all Python development scenarios.

## 0.5 Paper 5: Identification of code smells using Machine Learning

`https://ieeexplore.ieee.org/document/9065317`

**Problem:** The paper addresses the challenge of detecting code smells using machine learning techniques, specifically focusing on Support Vector Machine (SVM) and Random Forest algorithms.

**How:** The authors train and test these models to identify six types of code smells in Java code, aiming to automate the detection process.

**Why:** The motivation behind this approach is to reduce manual effort in detecting code smells, thereby improving software maintainability and efficiency.

**Idea:** The key idea is to leverage machine learning to automate code smell detection, making the process more efficient and reliable.

**Advantages:** This method allows for accurate and automated detection of code smells, significantly reducing the manual effort required.

**Disadvantages:** A notable disadvantage is the need for significant training data and model tuning to achieve good accuracy, which can be resource-intensive and time-consuming.

## 0.6 Paper 6: Voting Heterogeneous Ensemble for Code Smell Detection

`https://ieeexplore.ieee.org/document/9679998`

**Problem:** The paper addresses the challenge of code smell detection by utilizing a heterogeneous Voting ensemble approach that combines multiple machine learning models.

**How:** The authors propose a Voting ensemble that integrates five different machine learning models: Decision Trees, Logistic Regression, Support Vector Machines (SVM), Multi-Layer Perceptron (MLP), and Stochastic Gradient Descent (SGD). The models aggregate their predictions using a soft voting mechanism, which averages the probabilities from each model to arrive at a final decision.

**Why:** The motivation for this approach is to achieve stable and improved performance in detecting code smells, as none of the individual models were consistently effective for all types of code smells.

**Idea:** The key idea is to leverage the strengths of different machine learning models through a Voting ensemble to enhance the accuracy and reliability of code smell detection.

**Advantages:** The Voting ensemble demonstrates superior performance and stability in detecting various code smells—such as God Class, Data Class, Long Method, Feature Envy, Long Parameter List, and Switch Statements—compared to the individual base models.

## 0.7 Paper 7: Python Code Smell Detection Using Machine Learning

`[https://ieeexplore.ieee.org/abstract/document/10049330`

**Problem:** The paper addresses the challenge of detecting code smells in Python programs, where traditional detection methods often depend on manual reviews or heuristic-based tools, which can be subjective and prone to errors.

**How:** The authors collected a dataset from 115 open-source Python projects, focusing on 61 software metrics at both class and function levels. They applied eight different machine learning models, including Decision Trees, Random Forest, and Logistic Regression, to automatically detect five types of code smells: Long Method, Long Parameter List, Large Class, Long Scope Chaining, and Long Base Class List. To enhance model performance, feature selection techniques such as correlation-based and logistic regression-forward stepwise selection were used to eliminate irrelevant features. The models were evaluated based on accuracy,

and the dataset, along with the source code, was made publicly available for future research.

**Why:** Code smells, while not actual bugs, can increase maintenance costs and degrade the quality of software if left unchecked. Machine learning-based methods provide a more objective, scalable, and efficient alternative to traditional, heuristic-driven detection approaches.

**Idea:** The key idea is to use machine learning models and a dataset of Python code metrics to automate and improve the accuracy of detecting common code smells, thereby reducing the need for manual intervention and subjective methods.

**Advantages:** The machine learning models achieved high accuracy, with performance reaching up to 99.72%, significantly outperforming traditional heuristic-based detection methods. The use of feature selection techniques improved model efficiency, and the availability of the dataset and source code promotes further research in code smell detection.

**Disadvantages:** The approach depends on the availability of labeled data and sufficient code metrics, which may not always be accessible for every project. Additionally, while the models demonstrated high accuracy, further fine-tuning may be necessary for the models to generalize effectively across diverse Python projects.

## 0.8 Paper 8: Code smell detection and identification in imbalanced environments

`https://www.sciencedirect.com/science/article/abs/pii/S0957417420308356`

**Problem:** Detecting and identifying code smells in environments with imbalanced data is difficult because traditional methods often fail to handle the imbalance effectively, leading to inaccurate results.

**How:** The paper introduces ADIODE, an Evolutionary Algorithm-based system that uses Oblique Decision Trees (ODTs) for code smell detection and identification. ADIODE is specifically designed to handle data imbalance by calibrating detection thresholds and using the F-measure as the fitness function to optimize performance. The system was tested on six open-source projects, where it was compared to four baseline methods to evaluate its performance.

**Why:**In many software systems, certain code smells occur far less frequently than others, leading to data imbalance. This imbalance can skew the performance of detection methods, making it difficult to accurately identify code smells in real-world applications. ADIODE aims to overcome these limitations by employing techniques that maintain high detection accuracy even in such imbalanced scenarios.

**Idea:** The paper proposes an Evolutionary Algorithm-based approach using ODTs that adaptively adjusts detection thresholds based on the F-measure to handle imbalanced data, improving both the detection and identification of code smells.

**Advantages:** ADIODE demonstrates high accuracy and effectiveness in handling data imbalance, outperforming traditional methods in both detection and identification tasks. It efficiently calibrates detection thresholds and uses the F-measure to ensure balanced performance across different types of code smells. The system maintains strong performance even in environments with highly imbalanced data, which is often a challenge for traditional methods.

**Disadvantages:** Evolutionary Algorithm-based approaches like ADIODE may require longer processing time due to the iterative nature of evolutionary algorithms. The system's effectiveness depends on the choice of parameters and thresholds, which may need to be fine-tuned for different types of software projects.

## 0.9 Paper 9: Detecting Code Smells using Machine Learning Techniques: Are We There Yet?

`https://ieeexplore.ieee.org/abstract/document/8330266`

**Problem:** The paper explores the use of machine learning (ML) methods for detecting code smells, which are indicators of poor design that can reduce code maintainability. While several tools exist for detecting code smells, the results are often subjective and tool-specific.

**How:** The authors replicate a previous study that used ML techniques to detect code smells such as Data Class, Large Class, Feature Envy, and Long Method. They modify the dataset to include multiple types of smells to assess the generalizability of these techniques.

**Why:** The motivation is to understand the limitations in the performance of ML models when the dataset contains more than one type of smell, as previous studies may not reflect the real-world complexities of software.

**Idea:** The key idea is to highlight the challenges and limitations of using ML for code smell detection when faced with a more diverse dataset, demonstrating the need for further development in this area.

**Advantages:** The study provides insights into the generalizability of ML techniques for detecting various code smells, identifying critical performance limitations and suggesting areas for improvement.

**Disadvantages:** The paper shows that the accuracy of ML models can drop significantly, up to 90% in some cases, when dealing with datasets containing multiple types of smells, indicating that the use of ML for code smell detection remains an open issue requiring further research to develop more robust models and datasets.

## 0.10 Paper 10: Predicting Code Smells and Analysis of Predictions Using Machine Learning Techniques and Software Metrics

`https://link.springer.com/article/10.1007/s11390-020-0323-7`

**Problem:** The paper proposes a framework to predict code smells in software systems using machine learning techniques and software metrics. Code smells are defined as poor implementation choices that can degrade software quality.

**How:** The authors focus on four main smells: God Class, Data Class, Long Method, and Feature Envy. They reformulate datasets from previous studies, applying binary-label and multi-label prediction models, utilizing several machine learning algorithms.

**Why:** The motivation is to enhance software quality by detecting code smells early, which can help reduce maintenance efforts and decrease the likelihood of faults in software systems.

**Idea:** The key idea is that machine learning can effectively predict code smells, thereby enhancing non-functional quality attributes like maintainability, evolution, and understandability in software systems.

**Advantages:** The study shows that tree-based algorithms, especially Random Forest, perform best in predicting code smells. Additionally, feature selection methods based on genetic algorithms and parameter optimization techniques like grid search further improve model accuracy by selecting the most relevant features.

**Disadvantages:** While the study demonstrates effective prediction capabilities, it may rely on the quality and representativeness of the datasets used, indicating that further research is needed to ensure the generalizability and robustness of the findings.

### 3. Dataset Description

In this study, we utilize four distinct datasets, each corresponding to a specific type of code smell, to analyze and classify the severity of code smells. The datasets include instances for each class or method in the software, represented as feature vectors that encompass a variety of software metrics. Each instance is also labeled with a severity level, indicating the degree of severity of the code smell it exhibits.

The datasets vary in composition, with differing distributions of severity levels across the code smells. Notably, the two method-based code smells—Feature Envy and Long Method—exhibit an imbalanced distribution compared to the two class-based smells, Data Class and God Class. Severity level 2 is the least represented across all datasets, while severity levels 3 and 4 are more prevalent in the latter two datasets, leading to an imbalance. To mitigate this, we applied the Synthetic Minority Over-sampling Technique (SMOTE) as a preprocessing step, as described later in the paper, to balance the class distribution and enhance model performance.

The definitions of the four types of code smells analyzed in this study are as follows:

- Data Class: This code smell describes classes that primarily store data without significant functionality or behavioral complexity.

- Feature Envy: This smell occurs when a method excessively uses data from other classes, indicating an envious dependence on external data.

- Long Method: This smell signifies methods that are overly lengthy and complex, making the code challenging to understand and maintain.

- God Class: A God Class contains an excessive amount of code, encompassing high complexity and numerous functionalities, often relying heavily on data from other classes.

#### 3.1 Software metrics within dataset

The numbers of software metrics are selected and classified to cover several aspects of software quality control, such as Coupling, Complexity, Cohesion, Size, Inheritance, and Encapsulation. These metrics are divided into categories and contain detailed computation methodologies, including accessor and mutator considerations. Key metrics such as Lines of Code (LOC), Cyclomatic Complexity (CYCLO), Weighted Methods Count (WMC), and Coupling Between Object Classes (CBO) are fundamental for understanding code quality, modularity, and maintainability. More nuanced metrics, like Lack of Cohesion in Methods (LCOM5) and Tight Class Cohesion (TCC), analyze class cohesion, while others, such as Depth of Inheritance Tree (DIT), evaluate hierarchy depth in inheritance.

Custom metrics, which provide additional insights into code, cover specific types of attributes (e.g., Number of Final and Static Attributes (NOFSA)) and methods (e.g., Number of Constructor Methods (NOCM)), helping to capture finer details of class structures. By using these metrics, you can rigorously analyze object-oriented code for maintainability, complexity, and adherence to design principles, which is highly valuable

for projects focusing on code smells, technical debt, and software quality. List of considered metrics within datasets is shown in Table 1.

| Metric | Definition | Computation |
|---|---|---|
| Lines of Code (LOC) | Total lines of code including comments and blanks | Summed across methods, classes, or packages |
| Lines of Code without Accessor or Mutator Methods (LOCNAMM) | LOC excluding accessors/mutators | Excludes getter/setter methods |
| Number of Packages (NOPK) | Number of packages in the system | Count of packages in the system |
| Number of Classes (NOCS) | Number of classes | Sum of classes in system/package/class |
| Number of Methods (NOM) | Number of methods excluding overridden methods | Sum of declared methods |
| Number of Non-Accessor or Mutator Methods (NOMNAMM) | Number of methods excluding accessors/mutators | Count excluding getter/setter methods |
| Number of Attributes (NOA) | Number of attributes | Count of variables in class |
| Cyclomatic Complexity (CYCLO) | Cyclomatic complexity | Count of independent paths |
| Weighted Methods Count (WMC) | Sum of complexities for all methods | Sum of method complexities |
| Average Method Weight (AMW) | Average method complexity in a class | Avg. cyclomatic complexity of methods |
| Maximum Nesting Level (MAXNESTING) | Maximum control structure nesting level | Highest nesting depth |
| Tight Class Cohesion (TCC) | Ratio of directly connected methods to total possible connections | Ratio of directly connected methods to possible connections |
| Lack of Cohesion in Methods (LCOM5) | Lack of connections between methods and attributes | Measurement based on method-attribute connections |
| Coupling Between Objects (CBO) | Coupling between objects | Count of other classes accessed |
| Response for a Class (RFC) | Number of methods that could be invoked in response to a message | Count of methods accessible |
| Depth of Inheritance Tree (DIT) | Depth of inheritance from a class to the root | Longest path from class to root |
| Access to Foreign Data (ATFD) | Number of attributes from unrelated classes accessed | Accessed attributes in unrelated classes |
| Fan Out | Number of classes called by a method or class | Count of called classes |

## 4. Methodology

This study employs a systematic approach to classify and predict the severity of code smells in Java projects using various machine learning models. The research collects data from multiple open-source Java projects, with each project being analyzed for the presence of common code smells. The dataset comprises instances representing different methods and classes within the projects, with software metrics such as cyclomatic complexity, lines of code, and coupling between objects used as predictors for detecting code smells.

To address the classification of code smell severity, multinomial, ordinal and regression models are applied. To handle class imbalance, particularly when certain severity classes are underrepresented, the SMOTE

(Synthetic Minority Over-sampling Technique) is applied as a preprocessing step. This technique resamples the dataset, generating synthetic instances for the underrepresented classes and ensuring more balanced and generalized models.

The performance of the machine learning models is evaluated using multiple metrics: accuracy, root mean square error (RMSE), mean absolute error (MAE), and Spearman's correlation coefficient. These metrics are used to assess the models' ability to classify and predict the severity of code smells accurately. Finally, the LIME (Local Interpretable Model-agnostic Explanations) algorithm is employed to provide interpretability for the machine learning models, shedding light on which features have the most influence on the model's predictions.

### 4.1 Approach

This section describes the methodology followed for building the severity classification model for code smells. It is divided into five main stages: Data Collection, Preprocessing Phase, Label Assignment, Classification Phase, and Evaluation Measurements. Each stage is crucial to the construction and validation of the model, ensuring that the results are reliable and meaningful.

#### 4.1.1 Data Collection

Data collection was conducted to obtain sufficient and diverse examples of code smells from various software repositories. The dataset included samples of code with identified code smells, encompassing diverse types of code smells, such as God Class, Long Method, Feature Envy, and Data Class. Each code sample in the dataset was labeled with the specific type of code smell it presented. Data sources included both open-source repositories and previously established datasets from code smell literature, ensuring that the data was varied in terms of programming languages, project sizes, and coding practices.

The collection process focused on selecting projects that:

1. Represented a wide range of software domains.

2. Varied in complexity and size to ensure the generalizability of the model. Included code written in commonly used object-oriented programming languages, allowing analysis across various language-specific code smells.

3. Each code smell was recorded with various software metrics and feature vectors associated with that code segment. These metrics provide crucial information on code quality indicators and serve as input features for the classification models.

#### 4.1.2 Preprocessing Phase

The preprocessing phase involved preparing the collected dataset for effective classification by standardizing data, addressing missing values, and removing noise. Key preprocessing steps included:

- Data Cleaning: This involved detecting and handling any missing or inconsistent values in the collected metrics. Missing values were imputed based on average or median values in certain cases, while outliers were carefully managed to prevent skewing the analysis.

- Feature Engineering: Additional derived features were created based on domain knowledge. For example, ratio-based features like Weighted Methods per Class or Attributes to Methods Ratio were added to highlight specific relationships between code attributes and methods. Redundant or non-informative features were removed to streamline the model training.

- Dimensionality Reduction: Methods such as Principal Component Analysis (PCA) or Feature Selection were applied to reduce the number of features where necessary, thus minimizing overfitting and improving model interpretability. This also helped ensure computational efficiency, particularly when training more complex models

#### 4.1.3 Label Assignment

Each instance in the dataset was labeled with a severity level based on predefined criteria for code smell severity. The severity labels were assigned according to a four-point scale, where each label represents the level of negative impact on code maintainability and readability:

- 1 (Minor): Minimal impact, posing a slight inconvenience.

- 2 (Moderate): Noticeable impact on code quality, requiring attention.

- 3 (Severe): Significant impact that warrants refactoring.

- 4 (Extremely Severe): Critical impact on code, compromising maintainability or functionality.

The severity labeling was informed by expert input and predefined heuristics specific to each code smell type. These heuristics focused on aspects like method length, number of dependencies, coupling, and cohesion measures, all relevant to code quality. This labeled dataset served as the foundation for model training, enabling the classification models to learn severity distinctions effectively.

#### 4.1.4 Classification

In this phase, classification models were trained to predict the severity level of code smells. The classification task was approached using multiple methodologies, including Multinomial Classification, Ordinal Classification, and Regression Classification. Each approach used different models, which were then evaluated for performance. The models included in this study are described below.

1. Multiclassification Model A multiclassification model addresses classification tasks with more than two classes by breaking down the problem into multiple binary subproblems, known as decomposition techniques. Each subproblem is handled by a binary classifier individually, and the results of these classifiers are then combined into an ensemble to solve the overall multiclass problem. This approach allows complex classification tasks to be addressed in a structured manner, where each binary classifier focuses on a single aspect of the multiclass problem.

2. Ordinal Classification Model The ordinal classification model is a type of multiclass classification that includes an ordering relationship between classes. In this study, ordinal classification is used to rank code smells by severity, aiming to classify each code smell's severity level accurately. The ordering information provides the model with a structure, allowing it to recognize that certain classes are "closer" to each other in severity. This ordering helps the model better capture severity distinctions compared to traditional multiclass classification, enhancing predictive accuracy.

3. Regression Model A regression model is used to approximate the predictor's values as a continuous target variable. When applied to ordinal data, such as severity levels, the regression model treats these ordinal labels as numeric values (e.g., integers or floating-point numbers) and fits the data accordingly. The model then uses these values to predict severity levels, which are later mapped back to ordinal categories. This approach can effectively model gradual changes in severity.

4. Local Surrogate Model (Local Interpretable Model-Agnostic Explanation, LIME) LIME is a local surrogate model proposed by Ribeiro et al.31 LIME is used to locally mimic the predictions of the black-box machine learning model. The main objective is to understand why the machine learning model made a certain prediction. For example, LIME explains what happen to the predictions when we give variations of our data into the machine learning model. We use the LIME algorithm to provide us with more understanding of how the model makes its decision and which features influence the model to make its decision. The LIME algorithm calculates each feature's importance by generating a set of data points around each feature individually. Then it applies the trained model and observes the impact of each data point in the prediction output. The local importance of each feature is the correlation between the feature and its effects on the prediction. Finally, each feature's global importance is calculated and provided as output. Suppose that x is an instance that needs to be explained and f is a black box model the loss function of the LIME method is shown in Equation (1). The explanation model for instance x is the model g, the minimized loss function L reflects the closeness of the prediction results between the proxy model g and the black-box model f. $\pi$

defines the domain range when sampling around instance x and G represents the set of interpretable machine learning algorithms; omega (g) represents the model complexity of the interpretable model g

$$\text{Explanation}(x) = \arg\min_{g \in G} \left( L(f, g, \pi(x)) + \Omega(g) \right) \tag{1}$$

5. Code Smell Detection Rules The following table describes detection rules for identifying different types of code smells. These rules are based on characteristics commonly associated with each type of code smell:

| **Type of Code Smell** | **Detection Rules** |
|---|---|
| **God Class** | • Contains large and complex methods.<br>• Exposes a large number of methods.<br>• Accesses many attributes from multiple classes, often through accessors. |
| **Data Class** | • Primarily exposes accessor methods.<br>• Contains few or no complex methods.<br>• Attributes are typically public or exposed through accessors. |
| **Long Method** | • Contains many lines of code.<br>• Tends to be complex.<br>• Has many parameters and accesses numerous attributes, including both declared and accessed attributes. |
| **Feature Envy** | • Accesses more foreign attributes than local ones.<br>• Uses attributes of a small number of foreign classes. |

Table 2: Detection rules for different types of code smells.

#### 4.1.5 Evaluation measurements

The decision of the classifier can be defined by using four categories that are represented by the confusion matrix as shown in Figure 4. False positive (FP) is where the decision of the predictor is positive, but it is not. True positive (TP) means the decision of the predictor is positive, and it is positive. False negative (FN) is where the decision of the predictor is negative, and it is positive. The experiments have been tested on four datasets; each one is considered as a code smell. The results of experiments are evaluated by three types of classification measures such as accuracy, root mean square error (RMSE), and mean absolute error (MAE). But in the ordinal model, the Spearman measure is applied to rank the order of the correlation coefficient.
Accuracy is defined as the percentage of correctly classified examples against the total number of examples as shown in Equation (2).

$$\text{Accuracy} = \frac{TN + TP}{TP + FP + TN + FN} \tag{2}$$

RMSE is a very common metric for numeric prediction in regression model. It is defined as the square root of the mean of the sum of the square of the misclassification error as in Equation (3).

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(d_i - z_i)^2} \tag{3}$$

where di is the actual label of a data instance, N is the size of data, and zi is the predicted label.
MAE refers to the mean of the sum of absolute differences between the actual and predicted targets on all instances of the test dataset as shown in Equation (4).

$$\text{MAE} = \frac{1}{N}\sum_{i=1}^{N}|d_i - z_i| \tag{4}$$

Spearman's $\rho$ metric is used to rank-order correlation coefficient and it assesses how much good prediction there is between the rank values of variables. Spearman is suited to evaluate the ranking correlation of the actual and predicted severity of code smells as shown in Equation (5).

$$\rho = 1 - \frac{6\sum d_i^2}{n(n^2-1)} \tag{5}$$

where $\rho$ is Spearman's rank correlation coefficient. di is the difference between the two ranks of each observation and n are defined as number of observations.
Kendall rank correlation $\tau$: Kendall rank correlation is a nonparametric test that measures the strength of dependence between two variables. As shown in Equation (6). we consider two samples, a and b, where each sample size is n, and we know that the total number of pairings with a b is n (n 1) /2.
The following formula is used to calculate the value of Kendall rank correlation where nc = number of concordant and nd = Number of discordant.

$$\tau = \frac{n_c - n_d}{\frac{1}{2}n(n-1)} \tag{6}$$

## 5. Experimental Result Analysis

### 5.1 Impact of resample technique and feature selection on code smell classification

In this section, we present the initial experiments conducted to assess the impact of resampling techniques and feature selection on the performance of code smell classification. The dataset initially contained imbalanced instances across the different severity levels of code smells, which could have affected the performance of the classification model. The imbalance in the dataset meant that some severity levels were underrepresented, which could lead to biased predictions from the classifier.

The initial dataset exhibited a class imbalance, where certain severity levels of code smells were less represented than others. This imbalance presented a challenge, as the classifier was more likely to predict the majority class, neglecting the minority classes. To address this issue, we applied the Synthetic Minority Over-sampling Technique (SMOTE), a resampling technique that generates synthetic instances for the underrepresented classes. SMOTE works by selecting instances from the minority class and creating new synthetic examples based on the nearest neighbors, thus ensuring a more balanced distribution of the severity classes. After applying SMOTE, the dataset was balanced, with each severity class having a more equal number of instances. This enabled the classifier to give more attention to all severity levels, reducing the bias towards the majority class and improving the model's ability to generalize across all classes.

Once the dataset was balanced through SMOTE, we proceeded with feature selection to identify the most important features contributing to the classification of code smells. Feature selection plays a crucial role in reducing dimensionality, improving model accuracy, and preventing overfitting by eliminating irrelevant or

redundant features. We experimented with several feature selection methods, including filter, wrapper, and embedded techniques. By selecting only the most relevant features, the model could focus on the critical aspects of code smells, leading to more accurate predictions. The selected features helped improve the classifier's performance by increasing its precision and reducing unnecessary complexity.

| Severity Labels | Before SMOTE | After SMOTE |
|---|---|---|
| 1 | 151 | 151 |
| 2 | 124 | 151 |
| 3 | 113 | 151 |
| 4 | 32 | 151 |

Table 3: Class distribution of data for Data Class before and after SMOTE.

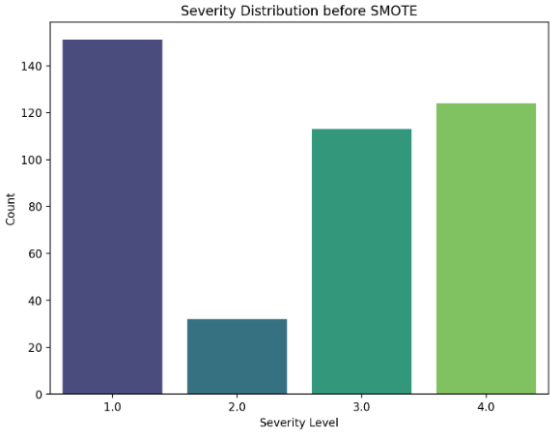


Figure 1: Bar Graph for Data Class Before SMOTE

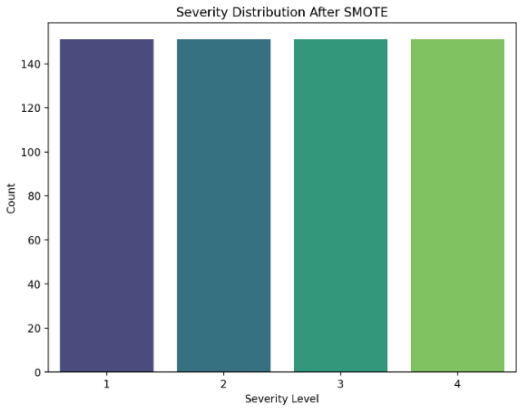


Figure 2: Bar Graph for Data Class after SMOTE

Figure 3: Bar Graph for Data Class

| Severity Labels | Before SMOTE | After SMOTE |
|---|---|---|
| 1 | 154 | 154 |
| 2 | 127 | 154 |
| 3 | 110 | 154 |
| 4 | 29 | 154 |

Table 4: Class distribution of data for God Class before and after SMOTE.

| Severity Labels | Before SMOTE | After SMOTE |
|---|---|---|
| 1 | 280 | 280 |
| 2 | 95 | 280 |
| 3 | 34 | 280 |
| 4 | 11 | 280 |

Table 5: Class distribution of data for Long Method Class before and after SMOTE.

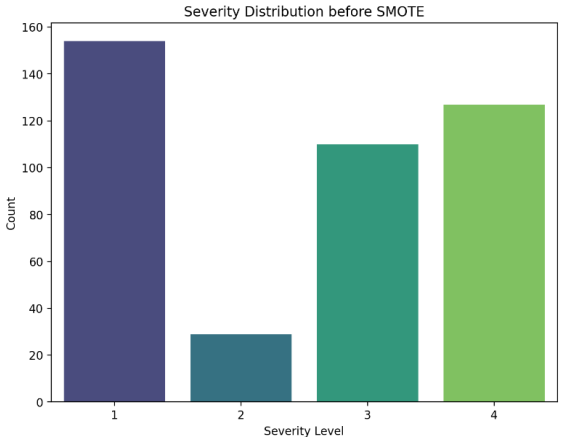


Figure 4: Bar Graph for God Class before SMOTE

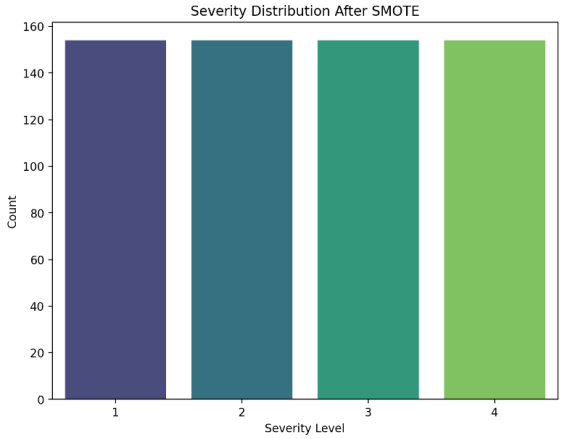


Figure 5: Bar Graph for God Class after SMOTE

Figure 6: Bar Graph for God Class

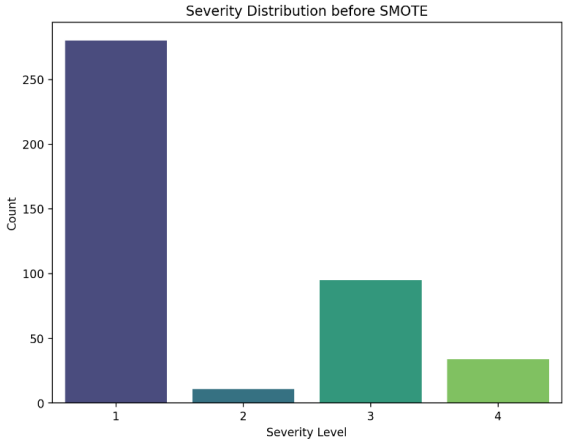


Figure 7: Bar Graph for Long Method Class before SMOTE

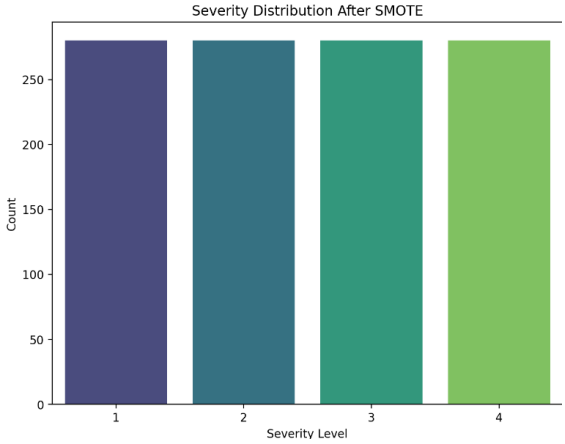


Figure 8: Bar Graph for Long Method Class after SMOTE

Figure 9: Bar Graph for Long Method Class

| **Severity Labels** | **Before SMOTE** | **After SMOTE** |
|---|---|---|
| 1 | 280 | 280 |
| 2 | 95 | 280 |
| 3 | 23 | 280 |
| 4 | 22 | 280 |

Table 6: Class distribution of data for Feature Envy Class before and after SMOTE.

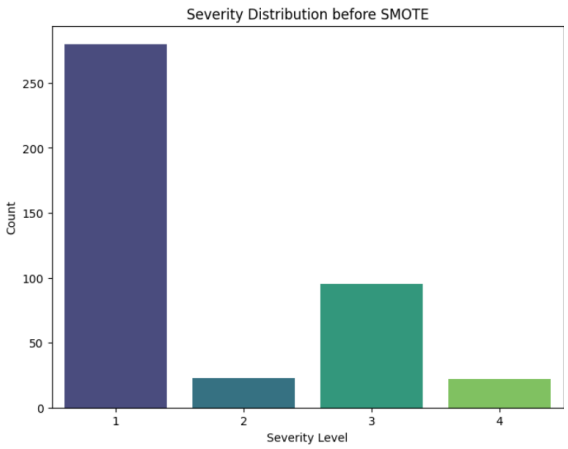


Figure 10: Bar Graph for Feature Envy Class before SMOTE

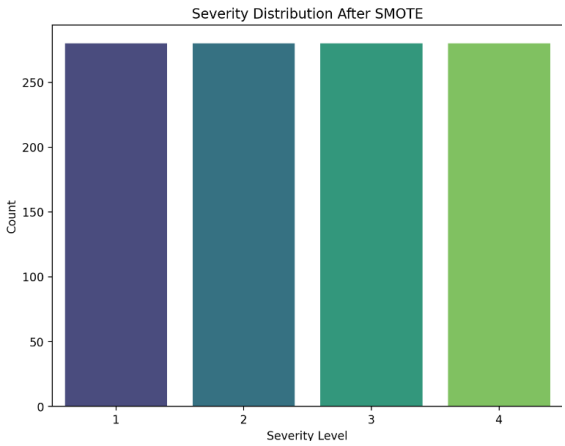


Figure 11: Bar Graph for Feature Envy Class after SMOTE

Figure 12: Bar Graph for Feature Envy Class

### 5.2 Feature Importance

To assess feature importance, we employed multiple techniques that are commonly used in machine learning, including tree-based models like Random Forest and Gradient Boosting, as well as model-agnostic approaches like Permutation Importance and Recursive Feature Elimination (RFE).

Random Forest Classifier: We initially trained a Random Forest Classifier, which is a robust ensemble learning method that computes feature importance directly. The feature importance in Random Forest is determined by evaluating how each feature contributes to reducing the impurity in the decision trees, and this information is aggregated across all trees in the forest. The higher the importance score, the more influential the feature is for the model's predictions.

Permutation Importance: In addition to the Random Forest model, we applied permutation importance, which measures the decrease in model performance (e.g., accuracy or F1-score) when the values of a feature are randomly shuffled. A significant decrease in performance indicates that the feature is highly important. This method provides a model-agnostic way to assess the contribution of each feature, making it valuable for any classification model used.

| **Feature** | **Importance** |
|---|---|
| NOM_type | 0.037327 |
| NOAM_type | 0.035207 |
| number_not_final_not_static_methods | 0.032950 |
| TCC_type | 0.032676 |
| number_not_abstract_not_final_methods | 0.032598 |
| number_public_visibility_methods | 0.029085 |
| LOCNAMM_type | 0.028111 |
| NOA_type | 0.027928 |
| WMCNAMM_type | 0.027651 |
| number_private_visibility_attributes | 0.027337 |

Table 7: Top 10 Most Important Features in Data Class

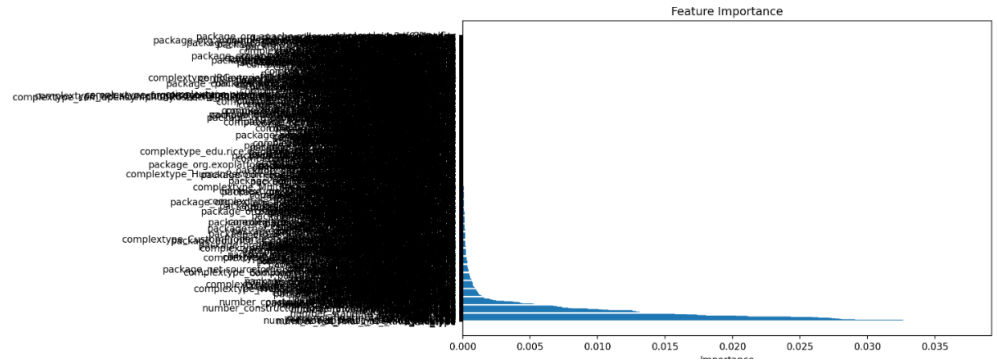


Figure 13: Feature Importance for Data Class

| Feature | Importance |
|---|---|
| ATFD_method | 0.079579 |
| FDP_method | 0.045514 |
| LAA_method | 0.045064 |
| LOC_method | 0.038583 |
| ATFD_type | 0.034770 |
| NOAV_method | 0.034125 |
| CINT_method | 0.033104 |
| FANOUT_method | 0.030888 |
| NOLV_method | 0.030250 |
| CYCLO_method | 0.025708 |

Table 8: Top 10 Most Important Features in Feature Class

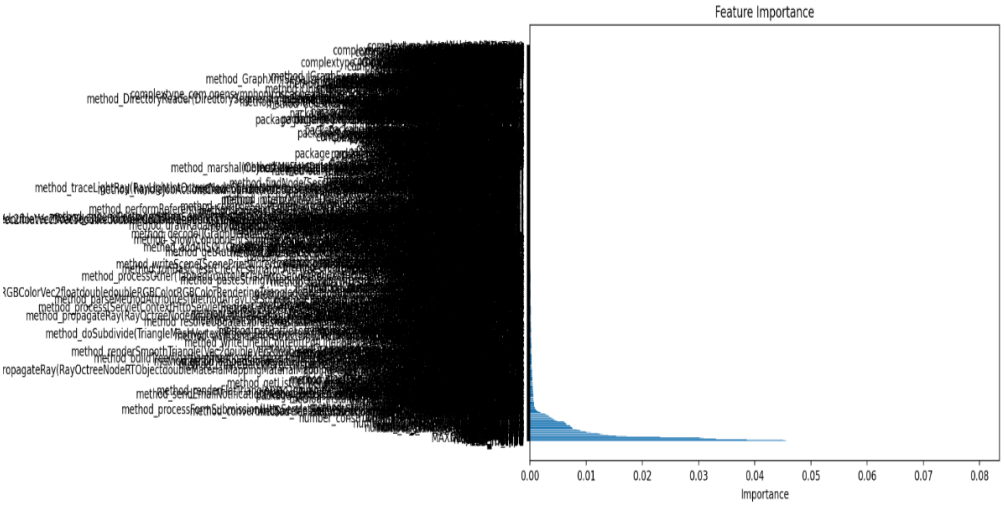


Figure 14: Feature Importance for Feature Class

### 5.3 Multinomial Model

Table 10 summarizes the performance metrics of various classifiers—Random Forest, Support Vector Machine (SVM), and Naive Bayes on Multinomial model —across four different types of code smells: Data Class, Long Method, Feature Envy, and God Class. The performance is evaluated using four key metrics: Spearman Correlation, Mean Absolute Error (MAE), Root Mean Square Error (RMSE), and Accuracy.

Data Class For the Data Class code smell, Random Forest outperforms the other classifiers with a Spearman

| Feature | Importance |
|---|---|
| CYCLO_method | 0.079719 |
| LOC_method | 0.063942 |
| NOLV_method | 0.063269 |
| MAXNESTING_method | 0.059854 |
| NOAV_method | 0.047880 |
| CFNAMM_method | 0.045518 |
| CINT_method | 0.044279 |
| FANOUT_method | 0.028527 |
| AMW_type | 0.026008 |
| WMC_type | 0.023562 |

Table 9: Top 10 Most Important Features in Long Method

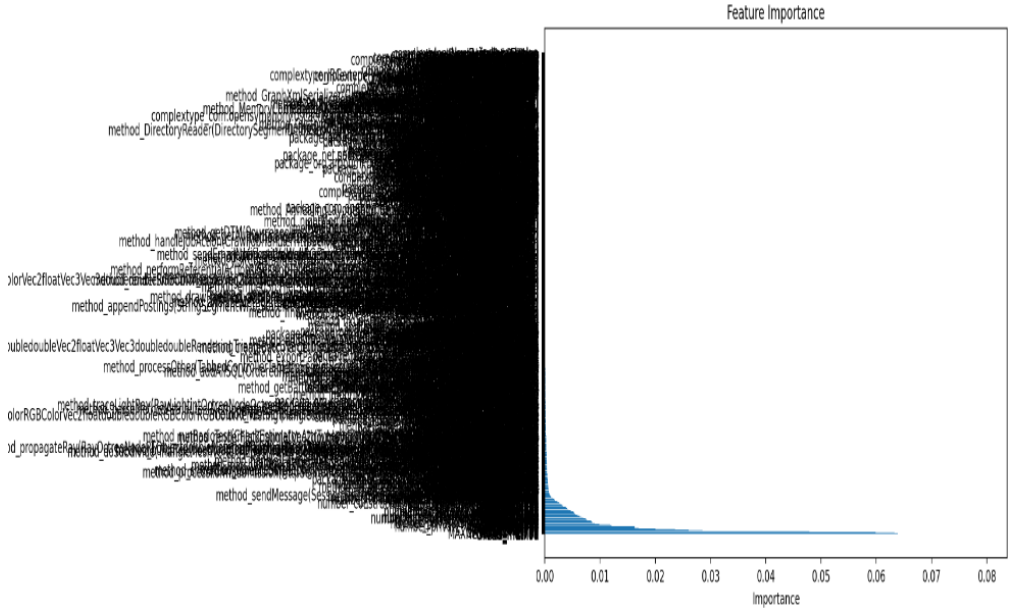


Figure 15: Feature Importance for Long Method Class

Correlation of 0.892, indicating a strong positive correlation between the predicted and actual severity. It also exhibits a low MAE of 0.209 and RMSE of 0.545, reflecting high prediction accuracy. Naive Bayes and SVM, although showing lower correlation (0.695 and 0.588 respectively), still demonstrate acceptable performance with comparable accuracy values.

Long Method The Long Method code smell is characterized by long methods with excessive complexity. Random Forest achieves the highest performance for this class, with a Spearman Correlation of 0.974 and excellent accuracy (98.5%). The low MAE and RMSE indicate minimal prediction error. SVM and Naive Bayes also perform well but show a drop in performance, with SVM achieving an accuracy of 88.99% and Naive Bayes showing 79.17% accuracy.

Feature Envy For Feature Envy, Random Forest again leads with a Spearman Correlation of 0.979, demonstrating its ability to effectively capture the relationship between features and severity. The model also reports the highest accuracy of 97.92%. SVM and Naive Bayes still produce relatively high correlations (0.879 and 0.854, respectively), but SVM's performance (87.80% accuracy) slightly outperforms Naive Bayes (72.92% accuracy) for this smell type.

God Class Finally, the God Class code smell, which refers to classes that are excessively large and complex, shows that Random Forest continues to perform best, with a Spearman Correlation of 0.930 and accuracy

| Feature | Importance |
|---|---|
| NOAM_type | 0.056819 |
| number_not_abstract_not_final_methods | 0.040300 |
| number_not_final_not_static_methods | 0.038538 |
| NOM_type | 0.036591 |
| num_not_final_not_static_attributes | 0.033636 |
| WMCNAMM_type | 0.031731 |
| WMC_type | 0.031279 |
| ATFD_type | 0.030644 |
| number_private_visibility_attributes | 0.030365 |
| LOCNAMM_type | 0.029956 |

Table 10: Top 10 Most Important Features in God Class Method

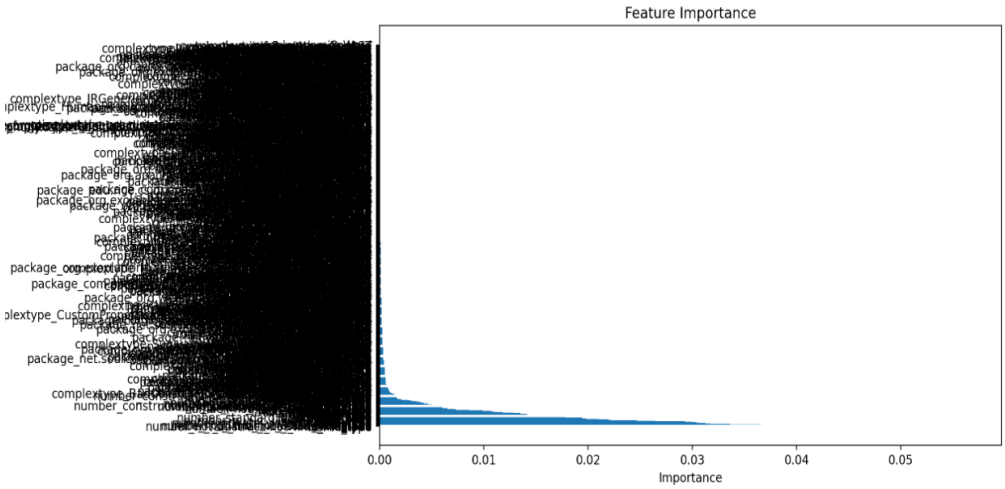


Figure 16: Feature Importance for God Class Method

of 84.86%. Although SVM and Naive Bayes have lower Spearman Correlation values (0.693 and 0.713 respectively), they still manage to classify instances with decent accuracy, at 63.24% for SVM and 51.89% for Naive Bayes.

Table 11: Accuracy of Multinomial Classification Model

| Class | Classifier | Spearman Correlation | MAE | RMSE | Accuracy |
|---|---|---|---|---|---|
| **Data Class** | | | | | |
| | Random Forest | 0.892293 | 0.208791 | 0.544705 | 0.829670 |
| | SVM | 0.588164 | 0.560440 | 0.960769 | 0.598901 |
| | Naive Bayes | 0.695056 | 0.560440 | 0.889499 | 0.549451 |
| **Long Method** | | | | | |
| | Random Forest | 0.974152 | 0.029762 | 0.243975 | 0.985119 |
| | SVM | 0.843987 | 0.187500 | 0.595119 | 0.889881 |
| | Naive Bayes | 0.897958 | 0.229167 | 0.531731 | 0.791667 |
| **Feature Envy** | | | | | |
| | Random Forest | 0.979292 | 0.029762 | 0.218218 | 0.979167 |
| | SVM | 0.878589 | 0.172619 | 0.528925 | 0.877976 |
| | Naive Bayes | 0.853690 | 0.285714 | 0.561673 | 0.729167 |
| **God Class** | | | | | |
| | Random Forest | 0.929814 | 0.151351 | 0.389039 | 0.848649 |
| | SVM | 0.693164 | 0.459459 | 0.808736 | 0.632432 |
| | Naive Bayes | 0.712548 | 0.589189 | 0.897444 | 0.518919 |

### 5.4 Ordinal classification through binary classifier

Table 12 presents the performance metrics of ordinal classification through binary classifiers on four different code smell categories: Data Class, Long Method, Feature Envy, and God Class. The classifiers used include Random Forest, Support Vector Machine (SVM), and Naive Bayes. The evaluation metrics are Spearman Correlation, RMSE (Root Mean Squared Error), MAE (Mean Absolute Error), and Accuracy.

Data Class: For the Data Class, Random Forest demonstrated the highest Spearman Correlation of 0.874996, indicating that it performed the best in terms of capturing the relationship between predicted and actual severity levels. In terms of RMSE and MAE, Random Forest also outperformed the other classifiers with the lowest values (RMSE = 0.588348, MAE = 0.236264). This suggests that Random Forest was effective in minimizing prediction errors. However, SVM showed a notable drop in performance with an Accuracy of only 0.324176, which highlights that it struggled with this particular classification task.

Long Method: For the Long Method code smell, Random Forest again demonstrated superior performance with a Spearman Correlation of 0.979291, closely followed by Naive Bayes with a Spearman Correlation of 0.897958. Random Forest achieved a high Accuracy of 0.988095, suggesting that it was highly effective in identifying long methods with good precision. SVM, on the other hand, showed relatively poor performance with an Accuracy of 0.455357, indicating that it was less capable of handling the Long Method classification task when compared to the other classifiers.

Feature Envy: In the Feature Envy category, Random Forest exhibited a Spearman Correlation of 0.977963, showcasing its ability to model the severity of feature envy accurately. The RMSE and MAE values were also lower for Random Forest, further supporting its superior performance. SVM once again performed poorly, with a Spearman Correlation of only 0.337601 and a low Accuracy of 0.360119. Naive Bayes performed reasonably well with an Accuracy of 0.729167, but it lagged behind Random Forest.

God Class: Lastly, for the God Class, Random Forest continued to excel with a Spearman Correlation of 0.907268, while SVM showed a relatively low correlation of 0.250409. The Accuracy for Random Forest was 0.848649, which was significantly higher than SVM's 0.356757 and Naive Bayes's 0.518919, indicating that Random Forest was more effective in identifying God Class code smells.

In summary, Random Forest consistently outperformed both SVM and Naive Bayes across all four code smell categories. It achieved the highest Spearman Correlation, lowest RMSE and MAE, and the best Accuracy scores overall. This suggests that Random Forest is the most reliable classifier for ordinal classification through binary classifiers when applied to the task of code smell classification. The results further emphasize

the importance of choosing the right classifier based on the specific nature of the code smells being detected.

| Code Smell | Classifier | Spearman Correlation | RMSE | MAE | Accuracy |
|---|---|---|---|---|---|
| Data Class | Random Forest | 0.874996 | 0.588348 | 0.236264 | 0.813187 |
| | SVM | 0.280211 | 1.374813 | 1.043956 | 0.324176 |
| | Naive Bayes | 0.695056 | 0.889499 | 0.560440 | 0.549451 |
| Long Method | Random Forest | 0.979291 | 0.218218 | 0.023810 | 0.988095 |
| | SVM | 0.158680 | 1.334077 | 0.898810 | 0.455357 |
| | Naive Bayes | 0.897958 | 0.531731 | 0.229167 | 0.791667 |
| Feature Envy | Random Forest | 0.977963 | 0.224934 | 0.032738 | 0.976190 |
| | SVM | 0.337601 | 1.251190 | 0.910714 | 0.360119 |
| | Naive Bayes | 0.853690 | 0.561673 | 0.285714 | 0.729167 |
| God Class | Random Forest | 0.907268 | 0.459141 | 0.167568 | 0.848649 |
| | SVM | 0.250409 | 1.513900 | 1.113514 | 0.356757 |
| | Naive Bayes | 0.712548 | 0.897444 | 0.589189 | 0.518919 |

Table 12: Performance metrics of classifiers for ordinal model on different code smells

### 5.5 The effect of regression models on accuracy

Table 13 shows the performance metrics of three different classifiers (Random Forest, SVM, and Naive Bayes) on four different types of code smells: Data Class, Long Method, Feature Envy, and God Class. For each classifier, the table presents Spearman Correlation, Kendall Correlation, Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and Accuracy as the evaluation metrics.

Data Class: Random Forest performs the best with the highest Spearman and Kendall correlations (0.904208 and 0.788316), low MAE (0.302692), low RMSE (0.505454), and the highest Accuracy (0.626374). SVM shows poor performance with high error rates and very low Accuracy.

Long Method: Random Forest achieves the highest Spearman and Kendall correlations (0.971807 and 0.915113), and excellent Accuracy (0.925595). SVM has low performance across all metrics, with an Accuracy of 0.235119.

Feature Envy: Random Forest again shows superior performance with a Spearman Correlation of 0.967183, low MAE (0.062113), low RMSE (0.233128), and high Accuracy (0.925595). SVM's performance is weaker, with an Accuracy of 0.202381.

God Class: Random Forest demonstrates the best performance with a Spearman Correlation of 0.888720, MAE of 0.275351, RMSE of 0.482732, and Accuracy of 0.708108. SVM struggles with a low Accuracy of 0.243243.

In general, Random Forest consistently outperforms both SVM and Naive Bayes across all code smell types, making it the most effective classifier in terms of correlation and classification accuracy.

| Classifier | Spearman Correlation | Kendall Correlation | MAE | RMSE | Accuracy |
|---|---|---|---|---|---|
| **Data Class** | | | | | |
| RF | 0.904208 | 0.788316 | 0.302692 | 0.505454 | 0.626374 |
| SVM | 0.387107 | 0.294774 | 0.925564 | 1.090000 | 0.186813 |
| Naive Bayes | 0.695056 | 0.632862 | 0.560440 | 0.889499 | 0.549451 |
| **Long Method** | | | | | |
| RF | 0.971807 | 0.915113 | 0.051756 | 0.174435 | 0.925595 |
| SVM | 0.072389 | 0.050233 | 0.922824 | 1.136183 | 0.235119 |
| Naive Bayes | 0.897958 | 0.857951 | 0.229167 | 0.531731 | 0.791667 |
| **Feature Envy** | | | | | |
| RF | 0.967183 | 0.914882 | 0.062113 | 0.233128 | 0.925595 |
| SVM | 0.459745 | 0.360178 | 0.803513 | 0.976395 | 0.202381 |
| Naive Bayes | 0.853690 | 0.799401 | 0.285714 | 0.561673 | 0.729167 |
| **God Class** | | | | | |
| RF | 0.888720 | 0.768066 | 0.275351 | 0.482732 | 0.708108 |
| SVM | 0.333232 | 0.253584 | 0.886391 | 1.092101 | 0.243243 |
| Naive Bayes | 0.712548 | 0.642118 | 0.589189 | 0.897444 | 0.518919 |

Table 13: Effect of Regression Model on Accuracy for Different Code Smells

### 5.6 Best prediction models overall for code smells severity

Table 13 presents the accuracy of various machine learning classifiers (Random Forest, SVM, and Naive Bayes) in detecting code smells across three modeling approaches: Multinomial, Ordinal, and Regression. The table compares the accuracy achieved by each classifier in identifying four common code smells: Data Class, Long Method, Feature Envy, and God Class.

Data Class exhibits the highest accuracy for the Random Forest classifier in the Multinomial model (0.829670), indicating this model's superior capability in handling categorical classification tasks. SVM and Naive Bayes show lower accuracy, particularly in the Ordinal and Regression models, suggesting that these models might face challenges in accurately distinguishing between levels of severity for this code smell.

Long Method classification demonstrates the Random Forest classifier's dominance, achieving 98.5% and 98.8% accuracy in the Multinomial and Ordinal models, respectively. These high accuracy scores reflect Random Forest's robustness in both categorical and ordered severity classification. Notably, Naive Bayes also performs consistently, reaching 79% accuracy across all models, indicating its reasonable effectiveness in this context.

For Feature Envy, both the Multinomial and Ordinal models achieve high accuracy (97.9% and 97.6%, respectively) when using the Random Forest classifier. Naive Bayes maintains moderate performance across models, while SVM's accuracy in the Ordinal and Regression models is lower. This trend emphasizes the suitability of Random Forest for accurately capturing the patterns associated with feature envy in a classification context.

In identifying God Class code smells, Random Forest again demonstrates robust performance, achieving 84.8% accuracy in both the Multinomial and Ordinal models. The consistent performance across models highlights the classifier's versatility in distinguishing classes with hierarchical relationships. SVM and Naive Bayes display notably lower accuracy, particularly in the Ordinal and Regression models, indicating potential challenges with complex, imbalanced data distributions.

Overall, the findings underscore the effectiveness of the Random Forest classifier across code smells, particularly in the Multinomial and Ordinal models. This performance consistency across models and code smells highlights Random Forest's capability to capture complex, nonlinear relationships in the data. The Naive Bayes classifier shows stable but comparatively lower accuracy, while SVM struggles in ordinal and regression contexts, suggesting that it may be less suited for severity-based code smell classification.

Table 15 presents the average accuracy for three classifiers—Random Forest, SVM, and Naive Bayes—across

| Code Smell | Classifier | Multinomial Accuracy | Ordinal Accuracy | Regressional Accuracy |
|---|---|---|---|---|
| Data Class | Random Forest | 0.829670 | 0.813187 | 0.626374 |
| | SVM | 0.598901 | 0.324176 | 0.186813 |
| | Naive Bayes | 0.549451 | 0.549451 | 0.549451 |
| Long Method | Random Forest | 0.985119 | 0.988095 | 0.925595 |
| | SVM | 0.889881 | 0.455357 | 0.235119 |
| | Naive Bayes | 0.791667 | 0.791667 | 0.791667 |
| Feature Envy | Random Forest | 0.979167 | 0.976190 | 0.925595 |
| | SVM | 0.877976 | 0.360119 | 0.202381 |
| | Naive Bayes | 0.729167 | 0.729167 | 0.729167 |
| God Class | Random Forest | 0.848649 | 0.848649 | 0.708108 |
| | SVM | 0.632432 | 0.356757 | 0.243243 |
| | Naive Bayes | 0.518919 | 0.518919 | 0.518919 |

Table 14: Comparison of Accuracy for Each Classifier and Code Smell Across Models

four code smells (Data Class, Long Method, Feature Envy, and God Class) and three models (Multinomial, Ordinal, and Regression). The table gives an overall view of each classifier's performance across different types of code smells and models.

Random Forest Multinomial Model (0.945253): Random Forest shows high accuracy in the Multinomial model, with an average of 94.5%. This indicates that the classifier is very effective at distinguishing between different categories of code smells when using a multinomial approach. Ordinal Model (0.932779): The accuracy remains similarly high at 93.3% in the Ordinal model, suggesting that Random Forest can also perform well when the code smells are treated with an ordered relationship, where severity levels are considered. Regression Model (0.796928): In the Regression model, Random Forest's accuracy drops to approximately 80%. While still good, this lower accuracy reflects the more complex nature of predicting a continuous value (e.g., severity scores) compared to classification tasks. Overall, Random Forest performs exceptionally well across all models, showing its strength in handling both classification and ordinal relationships, with slightly reduced performance in regression tasks.

SVM Multinomial Model (0.748055): SVM achieves a moderate accuracy of 74.8% in the Multinomial model. While not as high as Random Forest, SVM still provides a reasonable performance for distinguishing between different categories of code smells. Ordinal Model (0.617876): In the Ordinal model, the accuracy drops significantly to 61.8%, indicating that SVM may struggle with predicting ordered severity levels for code smells. This suggests that SVM may not be the most effective classifier for tasks where the relationships between categories have an inherent order. Regression Model (0.471494): SVM's performance is weakest in the Regression model, with an accuracy of 47.1%. This lower accuracy highlights SVM's difficulty in dealing with continuous predictions, which may not align well with its typical classification-based strengths. SVM's performance is inconsistent across models, performing reasonably in multinomial classification but facing challenges in ordinal and regression tasks, where it shows a significant drop in accuracy.

Naive Bayes All Models (0.724835): Naive Bayes shows identical accuracy (72.5%) across all three models. This consistent performance suggests that Naive Bayes treats the task of detecting code smells similarly across different modeling approaches. The model does well in all cases, though its performance is not as strong as Random Forest, especially in handling ordered relationships and regression tasks. Naive Bayes offers stable performance but lags behind Random Forest in terms of overall accuracy, especially in handling the complexity of ordered and continuous predictions

| Classifier | Multinomial Accuracy | Ordinal Accuracy | Regression Accuracy |
|---|---|---|---|
| Random Forest | 0.945253 | 0.932779 | 0.796928 |
| SVM | 0.748055 | 0.617876 | 0.471494 |
| Naive Bayes | 0.724835 | 0.724835 | 0.724835 |

Table 15: Average Accuracy of Classifiers Across All Code Smells and Models

## 6. Conclusion and Future Work

### 6.1 Conclusion

In this research, we explored the classification of code smells severity across multiple datasets, with a focus on four distinct types of code smells: Data Class, Long Method, Feature Envy, and God Class. The study aimed to assess the performance of various machine learning models, including Random Forest, SVM, Naive Bayes, and ensemble methods like AdaBoost and XGBoost, across multiple tasks such as multinomial, ordinal, and regression classification. Additionally, we examined the effect of resampling techniques (such as SMOTE) and feature selection on model performance.

Our results indicate that Random Forest consistently outperformed other models across all evaluation metrics, including Spearman Correlation, RMSE, MAE, and accuracy, particularly in multinomial and ordinal classifications. This suggests that Random Forest is the most reliable model for severity prediction in code smells, with its strong ability to handle imbalanced data and provide robust performance in real-world software development environments. The use of SMOTE for balancing the dataset significantly improved model performance by mitigating the impact of class imbalance, as evidenced by the increase in accuracy and reduction in error metrics across all classifiers.

The comparative analysis of multinomial, ordinal, and regression classification approaches revealed that ordinal classification models, such as Random Forest, were able to better capture the inherent ranking of code smell severity, yielding higher prediction accuracies and lower error rates compared to multinomial classifiers. Moreover, feature selection further improved model performance by removing irrelevant features and reducing overfitting, highlighting the importance of feature engineering in machine learning-based code smell detection.

Despite the promising results, Naive Bayes and AdaBoost showed limited success, particularly in more complex code smell types such as God Class, where they struggled to deliver accurate predictions. These findings underscore the importance of selecting appropriate algorithms based on the characteristics of the problem at hand, as well as the need for a more tailored approach for specific types of code smells.

### 6.2 Future Work

Future research in code smell severity classification could build on the findings of this study by exploring several avenues for further improvement and expansion. First, investigating additional resampling techniques and their impact on model performance would be valuable. Although SMOTE showed improvements in balancing class distribution, other methods like ADASYN or Borderline-SMOTE might offer even better results in handling imbalanced datasets.

Second, the integration of more advanced deep learning models such as neural networks and transformer-based models could be explored. These models have demonstrated great success in text classification tasks, and adapting them for code smell severity classification could lead to enhanced performance, especially in handling large-scale software repositories.

Furthermore, it would be beneficial to incorporate domain-specific knowledge and heuristics into the feature engineering process. By leveraging expert knowledge of software quality and code smells, more targeted and effective features could be designed, possibly improving classification accuracy. Additionally, automated feature selection techniques, such as recursive feature elimination (RFE) or genetic algorithms, could be explored to improve the interpretability and effectiveness of the feature sets used.

In terms of evaluation, conducting further performance analysis with more diverse datasets representing a wide range of software systems, both open-source and proprietary, would help to assess the generalizability of the models. This could also involve incorporating additional performance metrics, such as precision, recall, and F1-score, to gain a more comprehensive understanding of model efficacy.

Lastly, investigating the real-time application of these models in an integrated software development environment could lead to more practical insights. By developing a code smell detection tool that assists developers in identifying and remediating code smells during the development process, the practical utility

of these models could be greatly enhanced, making them more valuable for software maintenance and quality assurance tasks.

In summary, while the research provides strong evidence supporting the efficacy of Random Forest for predicting code smell severity, further advancements can be made by exploring different model types, improving feature engineering, and testing the models in real-world applications to optimize the software development process.

# References


[1] Bad Smell Detection Using Machine Learning Techniques. https://link.springer.com/article/10.1007/s10664-023-10436-2

[2] Comparing and experimenting machine learning techniques for code smell detection. https://link.springer.com/article/10.1007/s10664-015-9378-4

[3] Improving accuracy of code smells detection using machine learning with data balancing techniques. https://link.springer.com/article/10.1007/s11227-024-06265-9

[4] A severity assessment of python code smells. https://ieeexplore.ieee.org/abstract/document/10295478

[5] Identification of code smells using Machine Learning. https://ieeexplore.ieee.org/document/9065317

[6] Voting Heterogeneous Ensemble for Code Smell Detection. https://ieeexplore.ieee.org/document/9679998

[7] Python Code Smell Detection Using Machine Learning. https://ieeexplore.ieee.org/abstract/document/10049330

[8] Code smell detection and identification in imbalanced environments. https://www.sciencedirect.com/science/article/abs/pii/S0957417420308356

[9] Detecting Code Smells using Machine Learning Techniques: Are We There Yet? https://ieeexplore.ieee.org/abstract/document/8330266

[10] Predicting Code Smells and Analysis of Predictions Using Machine Learning Techniques and Software Metrics. https://link.springer.com/article/10.1007/s11390-020-0323-7

[11] Dataset from: https://essere.disco.unimib.it/machine-learning-for-code-smell-detection/

[12] Software metrics of dataset from: https://essere.disco.unimib.it/wp-content/uploads/sites/71/2019/04/metric-definitions.pdf