Severity Classification of Code Smells

Sanya Garg
Computer Science and Engineering
Netaji Subhas University of Technology
Delhi, India
sanya.garg.ug22@nsut.ac.in

Kairavi Kumar
Computer Science and Engineering
Netaji Subhas University of Technology
Delhi, India
kairavi.kumar.ug22@nsut.ac.in

Anurag Agarwal
Computer Science and Engineering
Netaji Subhas University of Technology
Delhi, India
anurag.agarwal.ug22@nsut.ac.in

Prof. Gaurav Singal
Asst. Prof. at Computer Science and Engineering
Netaji Subhas University of Technology
Delhi, India
gaurav.singal@nsut.ac.in

***Abstract*—Code smell refers to any characteristic of a code base that indicates a potential problem or flaw, often implying that the code is not optimal, maintainable, or readable. While code smells are not bugs or errors, they can lead to issues over time, such as making the code harder to understand, modify, and extend. 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 considerable research on identifying flaws in design patterns, less attention has been given to evaluating the severity of code smells or determining which machine learning models are most effective for this task.This study seeks to bridge this gap by applying various machine learning approaches, such as regression, multinomial, and ordinal classification models, to categorize the severity levels of code smells. Additionally, the LIME algorithm is employed to enhance the interpretability of the model predictions.**

***Index Terms*—Code smell, LIME, sampling, regression models, severity, multinomial, ordinal, random forest, naïve Bayes,XGBoost, SMOTE**

I. Introduction

In the world of software development, code quality plays a critical role in maintaining the longevity and reliability of a software system. As software grows in complexity and scale, developers face the challenge of ensuring that the code remains maintainable, efficient, and scalable. One of the key indicators of potential problems in the code is the presence of "code smells"—subtle signs that the code structure may need improvement. Identifying and addressing these code smells early in the development process can significantly improve the software's maintainability and reduce the risk of future issues.

A. 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.

B. 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.

C. Motivation

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.

D. 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 us identify which approach is most effective and contributes 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.

II. Literature/Related Work

A. Bad Smell Detection Using Machine Learning Techniques

Problem: The research addresses the difficulty of identifying code smells—signals of design problems that impact software maintainability, scalability, and clarity—traditionally discovered through time-consuming manual techniques. The potential role of automated methods in enhancing this detection process has not been fully explored.

How: The researchers conducted a thorough review of 17 studies to evaluate the effectiveness of various machine learning (ML) techniques for improving code smell identification. They trained models using datasets of known code smells and explored algorithms such as Support Vector Machines (SVM), J48 decision trees, and Random Forests with tools like WEKA.

Why: The aim was to create dependable, scalable tools for automated detection of code smells, enabling early refactoring and improving overall software quality. Machine learning approaches were examined for their ability to enhance accuracy and minimize human biases during detection.

Idea: The core concept is that ML can automate the identification of code smells, improving both the precision and speed of detection, while reducing the reliance on outdated and inefficient manual methods.

Advantages: The study concluded that machine learning, especially algorithms like J48 and Random Forests, significantly boosts both accuracy and speed in detecting code smells. It is efficient with large datasets and minimizes human biases during the detection process.

Disadvantages: Some obstacles include inconsistent datasets and limited scalability, with accuracy varying across different types of code smells. The use of ensemble learning techniques did not consistently lead to better results.

B. Comparing and experimenting machine learning techniques for code smell detection

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, 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 ML can automate the detection of code smells, thereby increasing detection precision 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.

C. Improving accuracy of code smells detection using machine learning with data balancing techniques

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.

D. A severity assessment of python code smells

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 categorize the intensity 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.

E. Identification of code smells using Machine Learning

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.

III. Dataset

In this paper, we use four distinct datasets, each of which corresponds to a specific type of code smell, to analyze and classify the severity of code smells. The datasets include instances for each class, represented as features 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 display varying compositions, with distinct distributions of severity levels among the identified code smells. Specifically, the two method-level code smells—Feature Envy and Long Method—show a more uneven distribution compared to the class-level smells, Data Class and God Class. Severity level 2 has the fewest occurrences across all datasets, whereas severity levels 3 and 4 are more prominent in the datasets corresponding to the class-level smells, resulting in an imbalance. To address this issue, we employed the SMOTE, a pre processing method detailed later in the paper, to equalize the class distribution and improve the performance of our models.

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

- Data Class: A class that primarily contains data without significant behavior or logic.
- Feature Envy: When a class frequently accesses and manipulates the data of another class, rather than its own.
- Long Method: A method that is excessively long and performs multiple tasks, making it difficult to understand.
- God Class: A class that has too many responsibilities and excessive control over other parts of the system.

A. *Software metrics within dataset*

These metrics are divided into categories and contain detailed computation methodologies, including accessor and mutator considerations. Key metrics such as Lines of Code, Cyclomatic Complexity, Weighted Methods Count, and Coupling Between Object Classes are fundamental for understanding code quality, modularity, and maintainability. More nuanced metrics, like Lack of Cohesion in Methods and Tight Class Cohesion, analyze class cohesion, while others, such as Depth of Inheritance Tree, evaluate hierarchy depth in inheritance.

Custom metrics, which provide additional insights into code, cover specific types of attributes and methods, 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.

TABLE I: Code Metrics and Computations

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

IV. 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 model accurately. Finally, the LIME 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.

A. *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.

a) 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, 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. The collection process focused on selecting projects that:

- Represented a wide range of software domains.
- 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.
- 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.

b) 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

c) 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) Sever: Significant impact that warrants refactoring.
4) Extremely Severe: Critical impact on code, compromising maintainability.

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.

d) Classification: In this phase, classification models were developed to forecast the intensity 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 multiclass classification model is designed to handle tasks involving more than two classes by dividing the problem into several binary classification subproblems, a method known as decomposition techniques. Each binary classifier addresses a specific subproblem, and the outputs from these individual classifiers are combined to form a solution for the overall multiclass problem. This method allows for the management of complex classification tasks in a structured way, with each binary classifier focusing on a distinct part of the multiclass issue.

2) Ordinal-Classification Model: An ordinal classification model is a distinct type of multiclass classification that takes into account a hierarchical relationship or sequence among the categories. In this research, it is employed to order code smells based on their intensity, with the objective of correctly assigning each code smell to its respective severity level. By integrating information about the order, the model is guided by a structured approach that acknowledges some classes are more similar in severity than others. This hierarchical perspective enhances the model's capacity to differentiate between severity levels, offering greater predictive precision compared to standard multiclass classification methods.

3) Regression Model: A regression model is designed to predict continuous values based on input data. When applied to ordinal data, such as severity levels, it treats these ordinal labels as numerical values (e.g., integers or real numbers) and fits the data accordingly. The model then uses these continuous values to make predictions about the severity levels, which are subsequently converted back into their original ordinal categories. This approach is effective for capturing gradual variations in severity.

4) LIME: LIME, is a local surrogate methodology aimed at interpreting the outputs of intricate, ML model. Its central goal is to shed light on the reasons behind specific predictions. For example, LIME facilitates the examination of how changes in input data influence the model's outputs. By applying the LIME algorithm, we can better understand the key factors driving the model's decision-making process.
The approach involves creating modified samples of data near a target instance and analyzing how these alterations impact the predictions. Feature significance is assessed by evaluating its effect on the model's behavior. Local importance highlights the role of a feature in a single prediction, whereas global importance aggregates the overall contribution of each feature across multiple instances.

Consider an instance x that requires interpretation, where f represents the ML model. The optimization process in the LIME framework is guided by a loss function, as expressed in Eq (1). The surrogate model providing an explanation for x is denoted by g. The loss function L quantifies the divergence between the predictions of the surrogate model g and the black-box model f.

The region for generating perturbed samples around x is denoted by π, while G represents the set of interpretable models. The complexity of the surrogate model g, indicated by $\omega(g)$, acts as a regularization term to ensure the model remains interpretable and simple.

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

5) Code Smell Detection: The 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:
   - **God Class:**
     – Contains large and complex methods.
     – Exposes a large number of methods.
     – Accesses many attributes from multiple classes, often through accessors.
   - **Data Class:**
     – Primarily exposes accessor methods.
     – Contains few or no complex methods.
     – Attributes are typically public or exposed through accessors.
   - **Long Method:**
     – Consists of a large number of lines of code.
     – Is often intricate and challenging to understand.
     – Includes numerous parameters and interacts with a wide range of attributes, both defined within and accessed externally.

- **Feature Envy:**
  - Relies more heavily on attributes from other classes than its own.
  - Primarily utilizes attributes from a limited number of external classes.

e) Evaluation measurements: The efficiency of the classification model is assessed through four distinct outcomes represented in the confusion matrix. A false positive occurs when the model incorrectly predicts a positive class for an instance that is actually negative. Conversely, a true positive signifies a scenario where the model correctly identifies an instance as belonging to the positive class. A false negative arises when the model incorrectly classifies a positive instance as negative.

The experiments were performed on four unique datasets, each associated with a specific type of code smell. The outcomes were analyzed using three evaluation metrics: accuracy,RMSE, and MAE. For the ordinal classification approach, the Spearman rank correlation coefficient was employed to evaluate the ranking consistency of the predictions.

Accuracy is determined as the ratio of correctly identified cases to the total instances in the dataset, as defined in Equation (2).

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

RMSE, is a widely used metric for evaluating numerical predictions in regression models. It is calculated as the square root of the average of the squared differences between the predicted and actual values, as represented in Equation (3).

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

Here, d_i denotes the true label of a data instance, N signifies the total number of instances, and z_i represents the predicted label for that instance.
The MAE as the mean of the absolute discrepancies between the true and predicted values across all instances in the test set, as presented in Equation (4).

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

The Spearman rank correlation coefficient ρ is employed to assess the degree of association between the rankings of variables. It quantifies how well the relative ordering of values in one variable aligns with the ordering in another.
This measure is especially effective for evaluating the consistency in ranking between the true and anticipated intensity levels of code smells, as represented in Eq (5).

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

Spearman's rank correlation coefficient (ρ) quantifies the degree of correlation between the ranks of two variables.
In this context, ρ is used to evaluate the alignment between the true and predicted ranks. Here, d_i represents the disparity between the rankings of each data point, and n denotes the total number of observations.

Kendall's rank correlation (τ) is a method that evaluates the degree of association between two variables without making assumptions about the underlying distributions.
As depicted in Equation (6), let a and b represent two samples, each containing n elements. The total number of pairs of elements is calculated as $\frac{n(n-1)}{2}$.
This is determined by n_c, the count of consistent pairs, and n_d, the count of inconsistent pairs, which are used to calculate τ.

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

V. Experimental Result Analysis

A. *Effect of Resampling Techniques on Code Smell*

In this section, we present the initial experiments aimed at assessing the effect of resampling techniques and feature selection. The dataset originally exhibited class imbalance, with certain severity levels of code smells being underrepresented. This imbalance could have resulted in biased predictions, as the classifier was more inclined to favor the majority class.

To address this issue, we implemented the SMOTE, which 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, thereby balancing the dataset and ensuring a more even representation of all severity levels. This improved the classifier's ability to generalize across all classes and reduced the bias toward the majority class.

Once the dataset was balanced, we performed feature selection to identify the most important features for code smell classification. Feature selection is crucial for reducing dimensionality, enhancing model accuracy, and preventing overfitting by removing irrelevant or redundant features. We tested various feature selection methods, including filter, wrapper, and embedded techniques. By selecting only the most relevant features, we enhanced the model's precision and reduced unnecessary complexity, resulting in more accurate predictions.

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

TABLE II: Data Class before and after SMOTE.

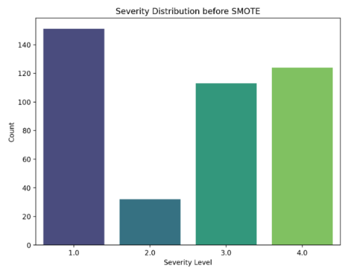

Fig. 1: Bar Graph for Data Class Before SMOTE.

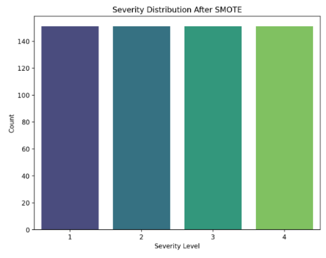

Fig. 2: Bar Graph for Data Class After SMOTE.

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

TABLE III: God Class before and after SMOTE.

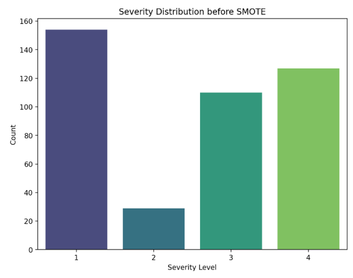

Fig. 3: Bar Graph for God Class Before SMOTE.

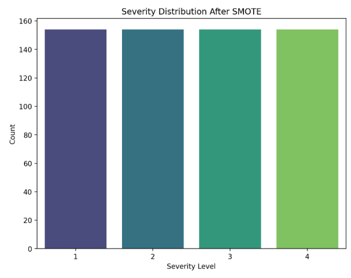

Fig. 4: Bar Graph for God Class After SMOTE.

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

TABLE IV: Long Method Class before and after SMOTE.

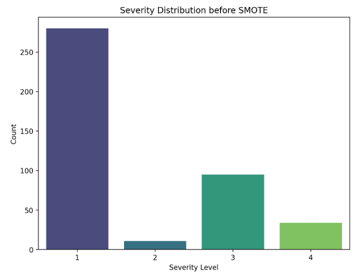

Fig. 5: Bar Graph for Long Method Before SMOTE.

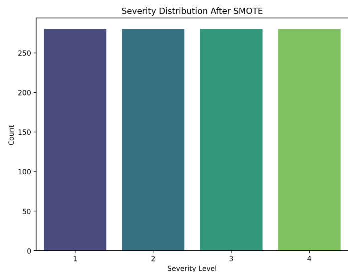

Fig. 6: Bar Graph for Long Method After SMOTE.

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

TABLE V: Feature Envy before and after SMOTE.

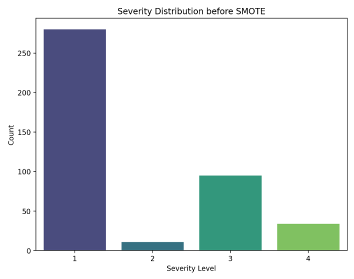

Fig. 7: Bar Graph for Feature Envy Before SMOTE.

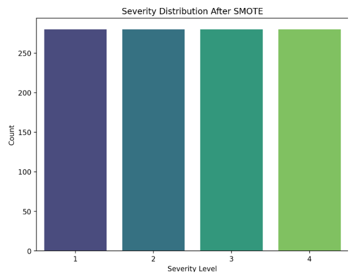

Fig. 8: Bar Graph for Feature Envy After SMOTE.

B. Feature Importance

To assess feature importance, we employed multiple techniques that are commonly used in machine learning, including 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. A higher importance score indicates that the feature has a stronger impact on the model's predictions.

Permutation Importance: We also used permutation importance, which evaluates the drop in model performance when a feature's values are randomly shuffled.(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 VI: Top 10 Most Important Features in Data Class

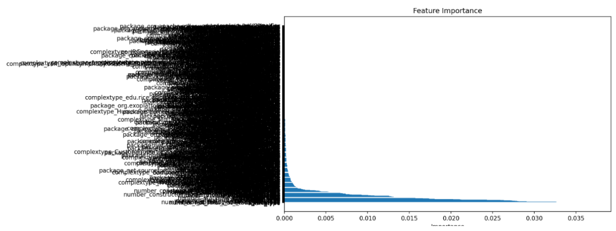

Fig. 9: Feature Importance for Data Class.

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 VII: Top 10 Most Important Features in God Class.

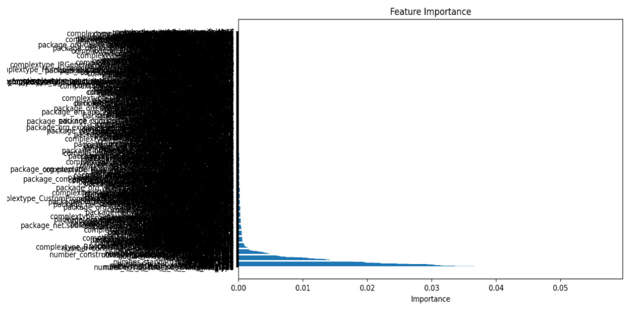

Fig. 10: Feature Importance for God Class.

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 VIII: Top 10 Most Important Features in Long Method

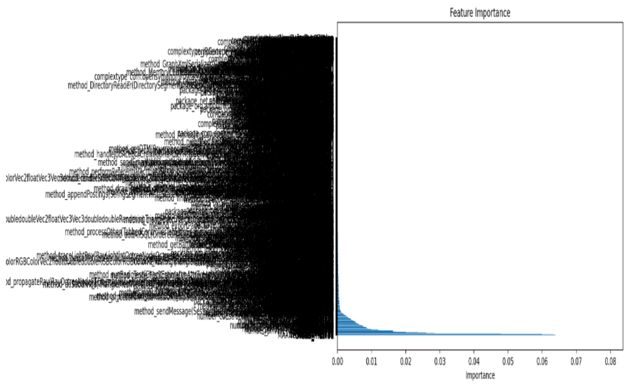

Fig. 11: Feature Importance for Long Method.

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 IX: Top 10 Most Important Features in Feature Envy.

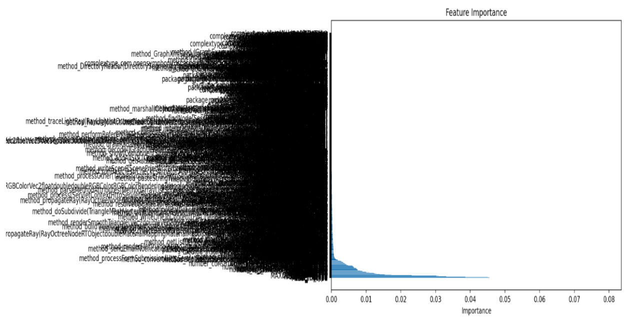

Fig. 12: Feature Importance for Feature Envy.

C. Multinomial Model Performance Summary

Data Class For the Data Class code smell, Random Forest achieves the best performance, with a Spearman Correlation of 0.892, indicating a strong positive relationship between predicted and actual severity. The model also shows a low MAE of 0.209 and RMSE of 0.545, suggesting high prediction accuracy. Although Naive Bayes and SVM have lower correlations (0.695 and 0.588, respectively), they still yield satisfactory results, with comparable accuracy scores.

Long Method In this case, which involves methods that are overly complicated and lengthy, Random Forest delivers the strongest performance, with a Spearman Correlation of 0.974 and an accuracy of 98.5%. The model's low MAE and RMSE indicate minimal error. SVM and Naive Bayes also show good results, although with lower accuracy: SVM reaches 88.99%, while Naive Bayes achieves 79.17%.

Feature Envy In the case of Feature Envy, Random Forest continues to lead with a Spearman Correlation of 0.979, demonstrating its strong ability to capture the link between features and their severity. The model reports the highest accuracy at 97.92%. SVM and Naive Bayes also perform well, with correlations of 0.879 and 0.854, respectively. However, SVM slightly outperforms Naive Bayes, achieving an accuracy of 87.80%, compared to Naive Bayes' 72.92%.

God Class For the God Class code smell, which relates to classes that are large and excessively complex, Random Forest remains the top performer, with a Spearman Correlation of 0.930 and an accuracy of 84.86%. Although SVM and Naive Bayes show lower Spearman Correlation values (0.693 and 0.713), they still manage to classify instances fairly accurately, with accuracies of 63.24% and 51.89%, respectively.

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

TABLE X: Accuracy of Multinomial Classification Model

D. Ordinal classification through binary classifier

Table 11 displays the performance metrics for ordinal classification using binary classifiers across the four code smells categories. The classifiers evaluated include Random Forest, Support Vector Machine, and Naive Bayes. The performance is assessed using the following metrics: Spearman Correlation, Root Mean Squared Error, Mean Absolute Error, and Accuracy.

Data Class: For the Data Class code smell, Random Forest showed the highest Spearman Correlation of 0.874996, which indicates its superior ability to capture the relationship between predicted and actual severity levels. Additionally, Random Forest outperformed other classifiers with the lowest RMSE (0.588348) and MAE (0.236264), reflecting its effectiveness in minimizing prediction errors. In contrast, SVM displayed a significant decline in performance, achieving an Accuracy of only 0.324176, indicating its difficulty in handling this classification task.

Long Method: For the Long Method code smell, Random Forest again demonstrated outstanding performance, with a Spearman Correlation of 0.979291, followed by Naive Bayes with a correlation of 0.897958. Random Forest achieved a high Accuracy of 0.988095, showing its capability in accurately identifying long methods. On the other hand, SVM performed poorly, with an Accuracy of just 0.455357, suggesting its limited ability to handle the Long Method classification compared to other classifiers.

Feature Envy: In the Feature Envy category, Random Forest recorded a Spearman Correlation of 0.977963, highlighting its ability to model the severity of feature envy effectively. The RMSE and MAE values for Random Forest were also lower, reinforcing its superior performance. SVM again struggled, with a low Spearman Correlation of 0.337601 and an Accuracy of 0.360119. Although Naive Bayes performed reasonably well with an Accuracy of 0.729167, it lagged behind Random Forest.

God Class: For the God Class code smell, Random Forest continued to outperform, achieving a Spearman Correlation of 0.907268. SVM, however, showed a much lower correlation of 0.250409. Random Forest's Accuracy of 0.848649 was notably higher than that of SVM (0.356757) and Naive Bayes (0.518919), indicating that Random Forest was far more effective in detecting God Class code smells.

In conclusion, Random Forest consistently outperformed both SVM and Naive Bayes across all four types of code smells. It achieved the highest Spearman Correlation, lowest RMSE and MAE, and the best Accuracy scores, making it the most reliable classifier for ordinal classification tasks related to code smell detection. These findings underline the significance of selecting the appropriate classifier based on the specific characteristics of the code smells being analyzed.

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

TABLE XI: Performance metrics for ordinal model

E. Impact of Regression Model

Table 12 summarizes the performance metrics of three classifiers—Random Forest, SVM, and Naive Bayes—across four code smell categories: Data Class, Long Method, Feature Envy, and God Class. For each classifier, the table reports the evaluation metrics: Spearman Correlation, Kendall Correlation, MAE, RMSE, and Accuracy.

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 XII: Impact of Regression Model

F. *Best Prediction Models for Code Smell Severity*

Table 13 displays the accuracy of different classifiers in detecting code smells across three modeling techniques: Multinomial, Ordinal, and Regression. The table compares the accuracy of each classifier in identifying four prevalent code smells.

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. The consistent performance across different models and code smells emphasizes Random Forest's ability to capture complex, non-linear patterns 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.

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

TABLE XIII: Comparison of Accuracy for Each Classifier

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 XIV: Average Accuracy of Classifiers Across All Code Smells and Models

VI. Conclusion and Future Work

A. Conclusion

This study investigated the classification of code smell severity across several datasets, focusing on four distinct categories of code smells: Data Class, Long Method, Feature Envy, and God Class. The objective was to evaluate the performance of various machine learning models, including Random Forest, SVM, Naive Bayes, and ensemble techniques like AdaBoost and XGBoost, across tasks such as multinomial, ordinal, and regression classification. Additionally, we explored the impact of resampling methods (like SMOTE) and feature selection on the models' performance.

The findings indicate that Random Forest consistently outperformed the other models across all evaluation criteria, including Spearman Correlation, RMSE, MAE, and accuracy, particularly in multinomial and ordinal tasks. This highlights Random Forest as the most reliable model for predicting the severity of code smells, demonstrating a strong ability to manage imbalanced datasets and deliver solid results in practical software development scenarios. The application of SMOTE to balance the data notably enhanced model performance, alleviating the effects of class imbalance, which was reflected in increased accuracy and decreased error metrics for all models.

The analysis comparing multinomial, ordinal, and regression classification approaches showed that ordinal models, particularly Random Forest, were better at capturing the ordered nature of code smell severity. These models achieved higher prediction accuracy and lower error rates compared to multinomial classifiers. Furthermore, feature selection contributed to improved model performance by eliminating irrelevant variables and mitigating overfitting, underscoring the significance of feature engineering in machine learning for code smell detection.

Although the results were promising, Naive Bayes and AdaBoost demonstrated limited effectiveness, especially for more complex code smell categories like God Class, where they struggled to make accurate predictions. These results emphasize the importance of choosing appropriate algorithms based on the problem characteristics and suggest that a more customized approach may be required for specific types of code smells.

B. Future Work

Future research in the classification of code smell severity could build upon the insights from this study by exploring several potential improvements and expansions. One promising direction is the examination of alternative resampling techniques to further enhance model performance. While SMOTE has shown benefits in addressing class imbalance, other approaches, such as ADASYN or Borderline-SMOTE, may provide even greater advantages in dealing with imbalanced datasets.

Another area for exploration is the application of advanced deep learning models, such as neural networks and transformer-based architectures. These models have achieved significant success in various text classification tasks and could be adapted for code smell severity classification, particularly in large-scale software repositories, leading to enhanced performance.

Additionally, incorporating domain-specific knowledge into the feature engineering process could yield valuable improvements. By utilizing expertise in software quality and code smells, more effective and targeted features could be developed, which may boost classification accuracy. Furthermore, automated feature selection methods like recursive feature elimination (RFE) or genetic algorithms could be investigated to refine the feature sets and improve their interpretability and overall model performance.

From an evaluation perspective, future work should involve a broader performance analysis across more diverse datasets, representing a variety of software systems, including both open-source and proprietary codebases. This would help assess the generalizability of the models. Incorporating additional evaluation metrics such as precision, recall, and F1-score would also provide a more comprehensive understanding of the models' effectiveness.

Finally, investigating the real-time application of these models within an integrated software development environment could offer valuable insights. Developing a code smell detection tool that assists developers in identifying and addressing code smells during the development process would enhance the practical utility of these models, making them more valuable for software maintenance and quality assurance tasks.

In conclusion, while this research demonstrates the efficacy of Random Forest in predicting code smell severity, future work could further improve performance by exploring alternative models, refining feature engineering, and testing the models in real-world scenarios, thereby optimizing the software development process.

References

[1] V. Pontillo, D. Amoroso d'Aragona, F. Pecorelli, D. Di Nucci, F. Ferrucci, and F. Palomba, "Machine learning-based test smell detection," *Software Quality Journal*, Accepted: 10 December 2023 / Published online: 5 March 2024.

[2] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 20, no. 4, pp. 1055–1086, June 2015.

[3] N. A. Khleel and K. Nehéz, "Improving accuracy of code smells detection using machine learning with data balancing techniques," *Journal of Computer Science and Technology*, Accepted: 21 May 2024 / Published online: 5 June 2024.

[4] A. Gupta, R. Gandhi, N. Jatana, D. Jatain, S. K. Panda, and J. V. N. Ramesh, "A severity assessment of Python code smells," *IEEE Access*, vol. 11, pp. 11312–11328, Oct. 2023.

[5] A. Jesudoss, S. Maneesha, and T. Lakshmi Naga Durga, "Identification of code smell using machine learning," *IEEE Xplore*, Added to IEEE Xplore: 16 April 2020.

[6] M. Y. Mhawish and M. Gupta, "Predicting Code Smells and Analysis of Predictions: Using Machine Learning Techniques and Software Metrics," Published: 30 Nov. 2020.

[7] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, "Detecting code smells using machine learning techniques: Are we there yet?," *IEEE Xplore*, Added: 5 Apr. 2018.

[8] S. Boutaib, S. Bechikh, F. Palomba, M. Elarbi, M. Makhlouf, and L. Ben Said, "Code smell detection and identification in imbalanced environments," Published: 15 Mar. 2021.

[9] N. Vatanapakorn, C. Soomlek, and P. Seresangtakul, "Python Code Smell Detection Using Machine Learning," Published: 24 Feb. 2023.

[10] H. Aljamaan, "Voting Heterogeneous Ensemble for Code Smell Detection," Published: 25 Jan. 2022.

[11] F. L. Caram, B. R. De Oliveira Rodrigues, A. S. Campanelli, and F. S. Parreiras, "Machine Learning Techniques for Code Smells Detection: A Systematic Mapping Study."

[12] F. A. Fontana, M. Zanoni, A. Marino, and M. V. Mäntyla, "Code Smell Detection: Towards a Machine Learning-Based Approach," Published: 2 Dec. 2013.