Prediction of Code Smells Severity For A Mobile Game

Gamze Tuncay
Dept of Computer Engineering
Istanbul Technical University
tuncay15@itu.edu.tr
504231518

***Abstract*—This paper presents a machine learning approach for detecting and classifying code smells at the class level in mobile game software. Using a combination of rule-based methods and machine learning algorithms, including Decision Trees and Random Forest, we classify code smells into low, medium, and high severity. The dataset, consisting of 136 classes, was analyzed with tools like SonarQube and Visual Studio Code Analysis. To address class imbalance, resampling techniques such as SMOTE and Random Oversampling were applied. The results show that while models performed well for low severity smells, they struggled with high severity instances due to data imbalance. This study highlights the potential of machine learning for automated code smell detection and suggests future improvements through expanded datasets and additional metrics.**

***Index Terms*—code smell, code quality, maintainability, machine learning, game development.**

I. Introduction

Poorly designed software is a significant problem that leads to various issues such as increased maintenance costs, reduced code reusability, and hardened debugging. Code smells, as defined by Beck and Fowler [1], are indicators of potential problems in the source code that refer to deeper issues in the underlying design. These smells, such as long methods, large classes, and god classes, can significantly affect the quality of the software. Code smell detection helps find problematic parts of code (like classes or methods) early on, even before testing [2]. This allows developers to plan maintenance and focus testing on the riskiest areas. By identifying and fixing code smells, developers can prevent future bugs and reduce the overall cost of the software.

Automated code smell detection tools can play a crucial role in identifying these issues early in the development life cycle, allowing developers to address them proactively. Machine learning (ML) is a popular choice for finding code smells because it can automatically discover hidden patterns in code data [3]. These algorithms can adapt to changes in the code and use a variety of information to identify potential problems. For example, classification algorithms can compare new code with existing code to see if it resembles code that has been identified as having a code smell. This helps developers find and fix problems early on, making the software easier to maintain.

This study focuses on finding and classifying the severity of code smells within class level. A system was developed that uses the ML method and the rule-based method. To do this, the code of a mobile game is analyzed using tools such as SonarQube [4] and Visual Studio Code Analysis [5], and the code metrics are extracted. Then, the rule-based technique is used to assign code smell severity levels. After the segmentation process, Decision Trees and Random Forest models are trained on this data. To improve the model's performance, Synthetic Minority Over-sampling Technique (SMOTE) and Random Over-Sampling is used to balance the dataset. Finally, the performance of our system is evaluated using accuracy, recall, and precision metrics.

This study contributes in several ways:

- New Code Smell Detection Model: A new model is developed specifically for identifying code smells at the class level within game software.
- ML Comparison: The performance of different ML techniques is compared in detecting these code smells for a mobile game.
- Resampling Effectiveness: The effectiveness of various resampling techniques is investigated in improving the performance of our machine learning models.

The rest of the paper is organized as follows. Section 2 presents related works about code smell. Section 3 gives details of used approach and background methodology. Section 4 presents data preparation and experimental results of the study. Lastly, Section 5 concludes the paper.

II. Related Works

Several different methods for identifying code smells have been described in research papers. Dewangan and Rao [6] proposed a study that aims to detect code smells from two datasets (Long Parameter List and Switch Statement) using ML methods, addressing class imbalance and selecting relevant metrics. They integrated SMOTE for class balancing and wrapper-based feature selection. Also, they utilized three ensemble ML techniques. The proposed methods demonstrates effective code smell detection. In other works of the authors [7] , they introduced a code smell detection approach by applying ensemble and deep learning models, utilizing the SMOTE technique for class balancing, and employing Chi-square feature selection to improve accuracy. The Chi-square feature extraction technique significantly improved the performance of the algorithms by identifying the most important features related to code smell detection. As the number of

selected metrics increased, the accuracy of nearly all ensemble methods improved, indicating that effective feature selection enhances model performance.

Khleel and Nehéz [8] presented a method that integrates various ML algorithms with a random oversampling technique to address class imbalance in datasets. The study evaluates the performance of five different ML models, reporting high accuracy rates for feature envy on original datasets, with improvements on balanced datasets. The results indicate that the combination of ML algorithms and data balancing significantly enhances code smell detection accuracy. This project was also motivated by the desire to understand how resampling methods can address the issue of class imbalance in our limited game project dataset.

Nizam et al. [9] introduces a novel model for detecting code smells at both line and block levels, utilizing machine learning and deep neural network (DNN) techniques. Key contributions include the development of a dataset from GitHub repositories, the application of various DNN methods, and a comparative analysis of ML and DNN performance, revealing that DNN methods outperform traditional approaches in understanding code structure. The results demonstrate an accuracy and F1 score between 0.75 and 0.80, underscoring the model's effectiveness in enhancing code quality and aiding software teams in identifying potential code smells.

Bosco et al. [10] presented UnityLint, a toolkit designed to detect code smells in Unity game development projects, based on an empirically derived catalog of 18 bad smells. The toolkit can be integrated into continuous integration pipelines for developers and serves as a research tool for studying Unity project quality, with future work planned to enhance detection rules and expand to other game development frameworks.

III. Experimental Setups and Methods

A variety of techniques is used to train a model to accurately identify code smell severity class (Fig 1). During the code metrics extraction process, SonarQube and Visual Studio were utilized. Metrics obtained from both tools were calculated at the class level. Highly isolated classes were excluded from the dataset.

The code smell metric calculated by SonarQube was selected for classification purposes. To analyze severity, the number of code smells was examined. Based on this, the segmentation was defined as low severity, medium severity, high severity. Additionally, if a class had a blocker issue that is also another metric provided by SonarQube, it was directly classified as high severity.

To evaluate the performance of different resampling, and ML algorithms, a series of experiments are conducted. The considered options are three different resampling options (SMOTE, Random Oversampling, and No Resampling), and two different ML algorithms (Random Forest, Decision Trees). This resulted in a total of 3 x 2 = 6 experiments. For each experiment, the data was divided into a training set and a test set. The resampling technique was applied to the training set. Then, the training set was utilized to train the ML algorithm,

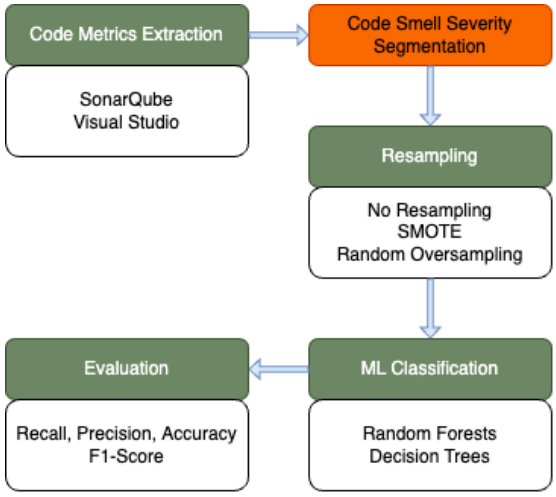

Fig. 1. The applied procedure

and its performance on the test set was assessed. This process was repeated for all 6 experiments.

The impacts of resampling, and feature selection techniques for each ML algorithms on the performance of the model was systematically evaluated through this experimental setup. The expectation is that the best combination of methods for the data set and task will be identified through this experiment. Also, the effectiveness of feature selection and sampling techniques with ML algorithms for imbalanced classification was investigated.

A. Dataset

The goal of this study is to develop a model that can accurately identify code smell severity. This is a challenging task, as the data set is limited and unbalanced. The dataset contains 136 classes, with 80 labeled as low severity, 41 labeled as medium severity, and 15 labeled as high severity. There are 14 metric used to train model which are listed in Table I. The first 8 metrics acquired from SonarQube and other 6 metrics acquired from Visual Studio Code Analysis.

B. Code Smell Severity Segmentation

For classification purposes, code smell severity segmentation were performed firstly. The code smell metric is defined by SonarQube as "the total count of code smell issues." The quantity of code smells was analyzed in order to do a severity analysis. Based on this, the classification was defined as follow Fig 2.

As shown in the Fig 2:

- If the number of code smells was lower than and equal to 2, it was categorized as low severity.
- If the number of code smells was between 3 and 8, it was categorized as medium severity.
- If the number of code smells exceeded 8, it was categorized as high severity.
- Additionally, if a class had a blocker issue that is also another metric provided by SonarQube, it was directly classified as high severity.

TABLE I
CODE METRICS

Metrics	Definition
Comment Lines	The count of lines that include comments or commented out code is measured, excluding non-significant comment lines such as empty comment lines or those containing only special characters.
Cyclomatic Complexity	A numerical measure that determines how many pathways there are in the code. The complexity counter is increased by one each time a function's control flow divides. The minimum complexity for each function is 1.
Cognitive Complexity	It evaluates how difficult it is to follow a code's control flow by combining Cyclomatic Complexity with human judgment. This approach provides method complexity scores that closely reflect developers' perceptions of maintainability, moving beyond purely mathematical models.
Technical Dept (Minute)	An attempt to address every code smell. The measure is kept in the database in minutes.
Reliability Remediation Effort (Mininute)	The effort to resolve every bug. The measure is kept in the database in minutes.
Functions	Number of functions in a class
Statements	Number of statements in a class
Duplicate Blocks	The number of duplicated blocks of lines.
Maintainability Index	The maintainability index has been redefined to range from 0 to 100. Originally, the metric was calculated using Halstead Volume, Cyclomatic Complexity and Lines of Code.
Depth of Inheritance	This refers to the longest path from a node to the root of the tree.
Member Count	Number of members in a class
Class Coupling	It also known as Coupling Between Objects (CBO), refers to the extent to which a class depends on other classes. A high coupling value is generally undesirable, while a low value is considered more favorable for this metric.
Lines of Source Code	This metric represents the total number of lines of source code in a file, including blank lines.
Lines of Executable Code	This metric represents the estimated number of executable code lines or operations, specifically counting the operations within the executable code.

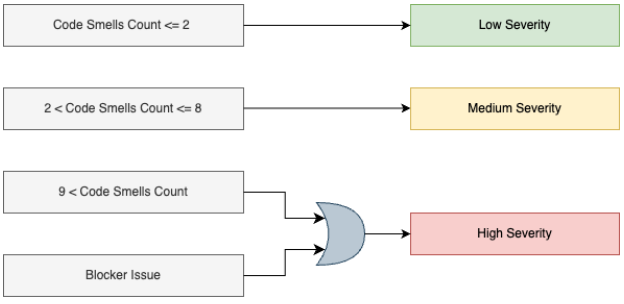

Fig. 2. Code smell severity tagging

C. Resampling

Resampling is a technique employed to equalize the number of instances per class, involving either under-sampling to decrease majority class instances or over-sampling to increase minority class instances [11]. Under-sampling randomly eliminates majority class instances, offering advantages such as improved run time and storage efficiency but potentially leading to biased remaining data and reduced accuracy for class distribution [12]. On the other hand, over-sampling randomly replicates minority class instances to achieve a balanced class distribution without eliminating data. While over-sampling avoids data loss and generally outperforms under-sampling, it may introduce over-fitting issues due to replicated instances [12].

1) SMOTE (Synthetic Minority Over-sampling Technique): It works by creating synthetic examples of the minority class. These synthetic examples are created by taking a minority class example and interpolating between it and its k nearest neighbors. This creates new examples that are located near the original minority class examples, but are not exactly the same. It also prevents the over-fitting issue from occurring during the training phase. Also, one of the most frequently used methods for addressing the problem of class imbalance in datasets is SMOTE [13].

2) Random Oversampling: In order to balance the dataset, random oversampling selects examples from the minority class at random and duplicates them [14]. It is computationally efficient and simple for implementation.

D. Machine Learning Classification

1) Random Forest: It is a powerful ensemble learning method and is a popular choice in ML due to its robust performance across diverse domains. Breiman (2001) introduced the concept of random forest as an ensemble of decision trees, where each tree is constructed using a random subset of features and trained on a bootstrap sample of the data. This ensemble approach mitigates over-fitting and enhances the model's generalization capability [15].

Random Forest operates through a process of bagging (Bootstrap Aggregating) and employs multiple decision trees, each trained on a different subset of the data set. The combination of individual trees' predictions, typically through a majority voting scheme, results in a more robust and accurate overall prediction. This ensemble technique leverages the diversity among the trees to enhance the model's performance [16].

Mathematically, the prediction of a Random Forest model can be represented as:

$$\hat{Y} = \text{mode}\{h_1(X), h_2(X), \ldots, h_n(X)\} \quad (1)$$

where $\hat{Y}$ is the predicted output, $h_i(X)$ represents the prediction of the i-th decision tree, and $mode$ mode denotes the mode function selecting the most frequent prediction.

The random subset of features used in each tree construction ensures de-correlation among the trees, contributing to the

TABLE II
MODEL RESULTS

Severity Level	Precision	Recall	F1-Score	Overall Accuracy
No Resampling- Random Forest				
Low	0.96	0.84	0.90	0.69
Medium	0.50	0.65	0.56	
High	0.00	0.00	0.00	
No Resampling- Decision Tree				
Low	0.94	0.91	0.92	0.76
Medium	0.59	0.76	0.67	
High	0.00	0.00	0.00	
SMOTE- Random Forest				
Low	1.00	0.84	0.92	0.73
Medium	0.55	0.71	0.62	
High	0.17	0.17	0.17	
SMOTE- Decision Tree				
Low	0.97	0.88	0.92	0.80
Medium	0.64	0.82	0.72	
High	0.50	0.33	0.40	
Random Oversampling- Random Forest				
Low	1.00	0.84	0.92	0.75
Medium	0.57	0.76	0.65	
High	0.20	0.17	0.18	
Random Oversampling- Decision Tree				
Low	0.96	0.84	0.90	0.78
Medium	0.61	0.82	0.70	
High	0.50	0.33	0.40	

model's stability and preventing it from being overly influenced by a single feature [17]. The Random Forest algorithm has been successfully applied in various fields, including classification, regression, and feature selection, making it a versatile and widely adopted tool in ML [17].

2) Decision Tree: Decision Tree is a supervised machine learning algorithm employed for both regression and classification tasks. It operates by dividing data instances based on feature values and expanding branches accordingly [8].In a typical decision tree, features are evaluated to identify the best splitting criteria, which helps maximize the homogeneity of data in the resulting branches. To determine the optimal splits, the algorithm calculates the Entropy of each feature. Entropy is a measure of impurity or disorder in the dataset. It quantifies the unpredictability of the data and ranges between 0 and 1. Entropy is crucial in guiding the tree's growth, as it helps identify the feature that best separates the data, thereby improving the tree's predictive accuracy [8].

IV. EVALUATION AND RESULTS

In this study, 48 classifier models based on Random Forest and Decision Tree are developed. To evaluate these models, 70% of the data set is used for training and validation while 30% is set aside for testing. The experimental results are shown in the Table II.

According to results we can conclude that:

- Most models, particularly Random Forest and Decision Tree, perform well in predicting low severity with high precision and good recall.
- While recall for medium severity is relatively good in some models (e.g., Decision Tree with SMOTE), precision is low, indicating a tendency for false positives and misclassification of medium severity instances.
- All models struggle significantly to predict high severity instances, with low or zero values for precision, recall, and F1-Score, likely due to class imbalance.
- Techniques like SMOTE and Random Oversampling improve recall for medium severity but do not significantly enhance performance for high severity, which remains difficult to predict.
- Decision Tree generally outperforms Random Forest in terms of precision and recall.

V. CONCLUSION

In this study, a ML model for detecting and classifying code smells at the class level in mobile game software was proposed. The model combined rule-based techniques with machine learning algorithms, such as Decision Trees and Random Forest, to classify code smells into low, medium, and high severity. Despite the challenges posed by class imbalance and a limited dataset, promising results were obtained, particularly for predicting low severity code smells.

However, the prediction of high severity code smells was found to be challenging due to the limited number of instances in this category. While resampling techniques such as SMOTE and Random Oversampling improved recall for medium severity smells, they did not significantly enhance performance for high severity.

It was observed that Decision Tree models outperformed Random Forest in terms of precision and recall, suggesting that simpler models may perform better in this context. Future work should focus on expanding the dataset to improve generalization across different game projects. Increasing the number of code metrics used for analysis could also potentially enhance the model's performance, especially in detecting high severity smells in more complex systems.

In conclusion, while the model demonstrated potential for code smell detection in mobile game software, further improvements can be made through data expansion, feature enhancement, and the refinement of classification techniques.

REFERENCES

[1] M. Fowler and K. Beck, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[2] F. Aktaş and F. Buzluca, "A learning-based bug predicition method for object-oriented systems," in *2018 IEEE/ACIS 17th International Conference on Computer and Information Science (ICIS)*, 2018, pp. 217–223.

[3] F. A. Fontana, M. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 21, pp. 1143 – 1191, 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:16222152

[4] SonarSource SA, "Sonarqube." [Online]. Available: https://www.sonarsource.com/

[5] Microsoft, "Visual Studio 2019." [Online]. Available: https://visualstudio.microsoft.com/

[6] S. Dewangan and R. S. Rao, "Method-level code smells detection using machine learning models," in *Computational Intelligence in Pattern Recognition*. Singapore: Springer Nature Singapore, 2023, pp. 77–86.

[7] S. Dewangan, R. S. Rao, A. Mishra, and M. Gupta, "Code smell detection using ensemble machine learning algorithms," *Applied Sciences*, vol. 12, no. 20, 2022. [Online]. Available: https://www.mdpi.com/2076-3417/12/20/10321

[8] N. Khleel and K. Nehéz, "Detection of code smells using machine learning techniques combined with data-balancing methods," *International Journal of Advances in Intelligent Informatics*, vol. 9, no. 3, pp. 402–417, 2023. [Online]. Available: http://ijain.org/index.php/IJAIN/article/view/981

[9] A. Nizam, M. Y. Avar, K. Adaş, and A. Yanık, "Detecting code smell with a deep learning system," in *2023 Innovations in Intelligent Systems and Applications Conference (ASYU)*, 2023, pp. 1–5.

[10] M. Bosco, P. Cavoto, A. Ungolo, B. A. Muse, F. Khomh, V. Nardone, and M. Di Penta, "Unitylint: A bad smell detector for unity," in *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*, 2023, pp. 186–190.

[11] E. F. Swana, W. Doorsamy, and P. Bokoro, "Tomek link and smote approaches for machine fault classification with an imbalanced dataset," *Sensors*, vol. 22, no. 9, p. 3246, Apr. 2022. [Online]. Available: http://dx.doi.org/10.3390/s22093246

[12] K. M. Hasib, M. S. Iqbal, F. M. Shah, J. Al Mahmud, M. H. Popel, M. I. H. Showrov, S. Ahmed, and O. Rahman, "A survey of methods for managing the classification and solution of data imbalance problem," *Journal of Computer Science*, vol. 16, no. 11, p. 1546–1557, Nov. 2020. [Online]. Available: http://dx.doi.org/10.3844/jcssp.2020.1546.1557

[13] D. Elreedy and A. F. Atiya, "A comprehensive analysis of synthetic minority oversampling technique (smote) for handling class imbalance," *Information Sciences*, vol. 505, pp. 32–64, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020025519306838

[14] R. Mohammed, J. Rawashdeh, and M. Abdullah, "Machine learning with oversampling and undersampling techniques: Overview study and experimental results," in *2020 11th International Conference on Information and Communication Systems (ICICS)*, 2020, pp. 243–248.

[15] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, Oct 2001. [Online]. Available: https://doi.org/10.1023/A:1010933404324

[16] A. Liaw and M. Wiener, "Classification and regression by randomforest," *Forest*, vol. 23, 11 2001.

[17] S. Xuan, G. Liu, Z. Li, L. Zheng, S. Wang, and C. Jiang, "Random forest for credit card fraud detection," in *2018 IEEE 15th International Conference on Networking, Sensing and Control (ICNSC)*, 2018, pp. 1–6.