Code smell severity classification at class and method level with a single manually labeled imbalanced dataset

Fábio do Rosario Santos
Military Institute of Engineering
Rio de Janeiro, RJ, Brazil
rosario.santos@ime.eb.br

Julio Cesar Duarte
Military Institute of Engineering
Rio de Janeiro, RJ, Brazil
duarte@ime.eb.br

Ricardo Choren
Military Institute of Engineering
Rio de Janeiro, RJ, Brazil
choren@ime.eb.br

ABSTRACT

Detecting code smells through machine learning (ML) poses challenges due to its unbalanced nature and potential interpretation bias. While previous studies focused on severity tended to categorize code smell's specific types, this research aims to detect and classify code smell severity in a single dataset containing instances of code smells of four distinct types: God-class, Data-Class, Feature-Envy, and Long-Method. This study also explores the impact of applying data scaling, feature selection techniques, and ensemble methods to enhance ML models for the purpose above. The evaluation of two ensemble models on a combined dataset reveals that using data standardization techniques, ensemble methods, and Chi-square outperforms the result of other ensemble combinations, achieving 81.04% and 81.41% accuracy in the XGBoost and CatBoost models. Additionally, the CatBoost algorithm attains the highest accuracy at 80.67%, even without data preprocessing. Comparatively with the state-of-the-art, the results obtained, an accuracy of 85%, by the proposed approach in detecting the severity of code smells are promising and suggest improvements in approaches and techniques to enhance the effectiveness and reliability of models in real-world scenarios.

KEYWORDS

code smells, severity, data preprocessing, feature selection, ensemble methods, machine learning

1 INTRODUCTION

Code smell detection is a data-driven process for code quality assurance that aims to detect whether a given piece of code violates fundamental design principles that negatively impact design quality [5]. Therefore, it proves valuable for software developers in minimizing maintenance costs and improving software quality by detecting code smells and their severity [4].

Several code smell detection tools, including commercial ones and research prototypes, have been proposed. These tools adopt diverse techniques to identify code smell: some are based on metrics [17, 21], while others employ a dedicated specification language [28], program analysis to identify refactoring opportunities [42, 43], analysis of software repositories [32], or machine learning (ML) techniques [5]. Most of these approaches, use program analysis to compare metrics against empirically identified thresholds, which can be biased [35]. The machine learning-based approach can explore developer-defined code characteristics, enhancing generalization across different datasets and programming languages [4].

However, detecting code smells using machine learning faces challenges inherent to its unbalanced nature and susceptibility to interpretation bias. This requires careful analysis and improvement of the internal mechanisms of the prediction model before interpreting the generated results [35]. The manually labeled dataset of the Madeyski Lewowski Code Quest (MLCQ) [23] is recognized as the most comprehensive one regarding different aspects such as sample size and quality [47]. Nevertheless, its code snippets have been labeled following a descriptive paradigm, which encourages subjectivity from annotators. In contrast, the prescriptive paradigm is data-centric and discourages this subjectivity, establishing clear criteria for annotating code smells in datasets [5, 29, 39].

Classifying the severity of code smells represents a crucial area of study, as it categorizes the problems associated with this domain, enabling the prioritization of costs and software maintenance efforts based on the severity faced, which can significantly contribute to the software life cycle [4]. In this context, this research work develops a model to detect and classify the severity of code smells through the combined use of data preprocessing, feature selection, hyperparameter optimization, and ensemble methods in a manually labeled imbalanced dataset.

This research area has recently grown in popularity within the research community, and, to address the problem of code smells, researchers have used various ML models. Initially, research focused only on code smell detection [5, 12, 14, 35], while recently, there has also been an increase in interest in investigating the severity of code smell [1, 4, 11, 18, 29, 37], intending to improve software quality.

Nevertheless, the aforementioned research studies, which center around severity, are primarily oriented toward categorizing the severity of code smells in datasets comprising instances of only a specific type of code smell. This orientation separates these approaches from real-world scenarios in existing software, as they neglect the influence exerted by different types of code smells on each other. In this sense, to verify the real capabilities of the code smell severity classification models evaluated, it is necessary to build a new dataset with instances of more than one severity type of different code smells, and a similar metric profile that may negatively affect the design quality [14].

This paper proposes an ML method for detecting and classifying the severity of a set of code smells at distinct levels: class and method. To accomplish this, first, we create a new code smell dataset from a combination of the datasets used in Arcelli Fontana and Zanoni [4] (two method-level datasets) and in Nanda and Chhabra [29] (two class-level datasets). The proposed dataset combines the observations from the four original datasets into a single observation in a new dataset, to present a more realistic representation of software code smells, since the software can present code smells from these two levels.

Following this, we present an ensemble method for the detection and severity classification of code smells at these two levels. To enhance the accuracy of the ensemble, the method addresses data scaling and feature selection. This allows for the detection of both non-smell and smelly code instances.

Thus, the contributions of this work can be described as two-fold:

- The creation of a dataset that contains instances of multiple types of code smell severities, which is more realistic and facilitates ML algorithms to better learn the nuances of these instances, based on a set of metrics, i.e., independent variables, encompassing several aspects of software quality such as size, complexity, cohesion, coupling, encapsulation, and inheritance.
- The proposal of a two-step approach to detect and classify code smell severity based on ensemble methods, that use data scaling and feature selection. Firstly, this approach allows the detection of instances with or without code smell following a binary approach. Furthermore, the classification of code smell severities is not restrained by the large number of negative instances, common in this problem type, as these instances are previously discarded in the second step of the approach, leaving only positive instances to be dealt with by the subsequently multiclass approach, which is more complex than the first.

To evaluate the proposed dataset and the ensemble method, we conducted an experiment based on the experiments conducted by Arcelli Fontana and Zanoni [4] and Nanda and Chhabra [29]. First, we apply data scaling (normalization and standardization) and feature selection (Linear Discriminant Analysis–LDA and chi-square) techniques separately to find out the impact of these techniques on the proposed dataset. Then we applied the technical improvements in combination with the ensemble methods (XGBoost and CatBoost) and compared them with state-of-the-art results. We discovered some promising results with our approach in classifying code smell severity at the class and method levels, since standardization achieved its highest accuracy in the CatBoost model, achieving a performance of 80.67%, and the combination of standardization techniques, chi-square, and CatBoost ensemble method achieved an accuracy of 81.41%. Finally, our two-step approach achieved relevant values (85%), surpassing the approach of Arcelli Fontana and Zanoni [4], while being outperformed by the approach of Nanda and Chhabra [29] (98%), in the context of a more realistic and complex dataset.

The remainder of the paper is organized as follows: section 2 presents related work. section 3 describes the process of combining the datasets. section 4 details the detection and severity classification ensemble. section 5 specifies the realized experiments and analysis obtained results. section 6 presents the threats to validity. Finally, section 7 describes the main conclusions of this paper.

2 RELATED WORK

This section presents studies that employed ML techniques to assess the severity of code smell. Also, some of these investigations examined the impacts of data preprocessing techniques, e.g., data balancing and feature selection techniques, and ensemble methods in the code smell severity classification. The benchmark dataset in these studies originated from research proposed by Arcelli Fontana and Zanoni [4].

The study of Arcelli Fontana and Zanoni [4] focused on evaluating the severity of code smell by applying ML techniques in several experiments. Different models were tested, from multinomial classification to regression, including a method to adapt binary classifications to ordinal classification. Notably, they modeled code smell severity as an ordinal variable, expanding on models proposed by Arcelli Fontana et al. [5], where promising results were achieved by employing binary classification models for code smell detection. However, they did not show the use of normalization or sophisticated feature selection approaches.

In a study by Nanda and Chhabra [29], a detailed analysis and correction of the datasets from the research of Arcelli Fontana and Zanoni [4] was carried out, removing inconsistencies in the data related to the God and Data classes. This intervention improved the performance of the ML techniques used for code smell severity classification. Later, they proposed an approach called Stacked Hybrid Model (SSHM) –Synthetic Minority Oversampling Technique (SMOTE) and Stacking in combination, applying it to the severity classification of the four types of code smell, namely: God-class, Data-class, Feature-envy, and Long-method. However, both studies [4, 29] did not consider other performance metrics, such as precision, recall, or F-measure, limiting the analysis of the results to a confusion matrix.

The study conducted by Abdou and Darwish [1] aimed to evaluate the performance of classification models for the severity of code smell after applying the SMOTE resampling technique. An additional focus of the study was the selection of the best models through a comparative analysis. Additionally, they extracted prediction rules to examine the effectiveness of using software metrics in code smell prediction. However, they did not use any data scaling or feature selection techniques in the data preprocessing phase of their approach.

Hu et al. [18] evaluated the performance of 21 prediction models in estimating code smell severity. These models included 10 classification methods and 11 regression algorithms, where the main performance metrics adopted were Cumulative Lift Chart (CLC) and Severity@20%. The latter is used to assess the percentage of the total severity of the predicted top 20% of software instances. In contrast, accuracy was used as a secondary indicator. Nevertheless, they did not use data preprocessing techniques before applying the code smell severity classification methods and algorithms.

In their research, Rao et al. [37] employed the SMOTE method to address the class imbalance challenge and applied the Principal Component Analysis (PCA)-based feature selection technique to improve accuracy. Conversely, Dewangan et al. [11] used ensemble ML models, employing a selection approach Chi-square features in each dataset, and evaluated the impact of parameter optimization on classification results for detecting the severity of code smell. Although both studies used the normalization technique for data scaling, this use was not accompanied by a comparison with other data scaling techniques, e.g., data standardization, nor the impact analysis of these techniques.

These research studies, focusing on severity, mainly aimed at classifying the severity of code smells in datasets that contained instances of only one specific type of code smell. This narrow focus

separates these approaches from real-world situations in actual software development. By doing so, they overlook the interplay and influence between different types of code smells on each other.

Unlike previous works, to achieve the goal of detecting and classifying code smells, we combine the two method-level datasets from Arcelli Fontana and Zanoni [4] and the two class-level datasets from Nanda and Chhabra [29] into a single one. This combination aims to more accurately represent the challenge encountered in real software, in which code smell instances, both at method and class levels, are influenced by distinct forms of code smell severity. We also propose a two-step approach to detect and classify code smell severity based on ensemble methods, that use data scaling and feature selection. This approach allows for the detection of instances with or without code smells, following a binary approach, and the classification of code smell severities, performed only with positive instances in a multiclass approach.

3 DATASET COMBINATION

The original datasets are available, respectively, at Evolution of Software Systems and Reverse Engineering (ESSeRE Lab)[1] –Arcelli Fontana and Zanoni [4] and in *Google Drive*[2] –Nanda and Chhabra [29]. For a clearer understanding, we present the following definitions for these code smells:

God-class (GC) refers to classes that centralize the system's intelligence and can tend to exhibit characteristics such as complexity, excess code, abundant use of data from other classes, and implementation of multiple functionalities [18].

Data-class (DC) are classes that mainly serve as data storage, they have limited functionality, and many other classes depend on them. These classes have multiple attributes, have no complexity, and provide access to data through accessor methods [18].

Feature-envy (FE) refers to methods that predominantly access and use data from other classes rather than their own. These methods are often characterized by excessive use of attributes from other classes, including those accessed through accessor methods [18].

Long-method (LM) is characterized by methods that encapsulate an excessive amount of functionality within a single class. These methods tend to be time-consuming, complex, difficult to understand, and rely extensively on data from other classes [18].

In their study, Arcelli Fontana and Zanoni [4] selected 76 systems out of 111, that were evaluated for different sizes, considering an extensive set of object-oriented resources. The choice of systems was based on Qualitas Corpus version 20120401r, collected by Tempero et al. [41]. To detect the code smell severity, they employed a variety of tools and methods known as advisors, such as iPlasma [24], Fluid Tool [30], JSpIRIT [45], PMD [24], and Marinescu detection rules [25]. Table 1 presents the automatic detection tools used in the study.

In the study conducted by Nanda and Chhabra [29], during the analysis of the datasets provided by Arcelli Fontana and Zanoni [4], a significant number of inconsistencies were identified between binary and multinomial datasets from GC and DC. These discrepancies were appropriately corrected, and the remaining instances were thoroughly evaluated for any misclassifications, requiring adjustments in severity where necessary. Several advisors were employed during this correction phase, notably iPlasma [24], JSpIRIT [45], and PMD [3] for GC, while the iPlasma tool was used for DC, and the final decision and severity labels were assigned based on expert assessment. Table 2 summarizes the different corrections made, indicating the reasons for each adjustment.

Table 1: Automatic detection tools (advisors)

Class	Advisors
DC	iPlasma
GC	iPlasma, JSpIRIT, PMD
FE	iPlasma, Fluid Tool
LM	iPlasma (Brain Method), PMD, Marinescu detection rules

Table 2: Details of corrected instances provided by Nanda and Chhabra [29]

Reason	God-Class	Data-Class
A. Conflict between dataset	128	129
B. Misclassified as negative instance	5	12
C. Misclassified as positive instance	6	6
D. Severity reclassification	18	20
E. Non-conflicting changes (B+C+D)	29	38
Total (A + E)	157	167

Each of these four datasets includes 420 instances, representing classes or methods. These instances are associated with a feature vector, composed of 63 values for GC and DC, and 84 values for LM and FE. Furthermore, each dataset contains information about the severity value for the code smell, as:

0 - No code smell: the class or method is not affected by code smell. A no LM example: a method with 10 lines of code is responsible for calculating the number of products sold. In its final line, the method also saves the result into a file, which is a different responsibility. Since the second responsibility is minor (i.e., one easy-to-understand line of code), it does not present an issue [39].

1 - Non-severe code smell: the class or method is partially affected by code smell. A non-severe LM example: a method with 30 lines of code has three distinct regions that perform different responsibilities. Each region can be extracted into a standalone method [39].

2 - Code smell: the characteristics of code smell are all present in the class or method. An LM example: a method has 100 lines of code and includes multiple levels of nested control structures. Refactoring such a method requires understanding the tasks it performs, reorganizing its logic, splitting loops, and extracting regions of cohesive logic [39].

3 - Severe code smell: the code smell is present and has particularly high values of size, complexity, or coupling. A severe LM example: a method with several hundred lines of code and many

[1]http://essere.disco.unimib.it/reverse/MLCSD.html
[2]https://drive.google.com/drive/folders/16BqUdNlKNgdM_qrrJqGWKdQ_NfEdRPVD?usp=sharing
[3]https://github.com/pmd/pmd

nested control structures. It is very difficult to comprehend, solves many tasks, and has nonapparent side-effects [39].

The dataset combination process was performed to make the dataset applicable to ML techniques in real-world scenarios where imbalanced data is common due to the inherent nature of code smells. In this sense, to verify the real capabilities of the code smell severity classification models evaluated, it is necessary to build a new dataset with instances of more than one severity type of different code smells, with a similar metric profile that may negatively affect the design quality. The feature vectors of the GC and DC datasets were modified to include the 84 values corresponding to FE and LM. This resulted in missing values, which are covered in subsubsection 5.1.1.

Figure 1 illustrates data from Arcelli Fontana and Zanoni [4] and Nanda and Chhabra [29], which initially contains 420 instances each. These instances are sorted in ascending order by attributes: "project", "package", "complex_type", and "method". In the first stage of the process, each pair of class-level (GC and DC) and method-level (FE, LM) datasets are combined. For this, the algorithm detailed in Algorithm 1 is presented. In brief terms, this algorithm deletes duplicate method- or class-level instances with different severities by removing the lowest severity. Furthermore, in the case of instances with equal severities, instances of the LM and GC classes are prioritized over FE and DC, respectively, since they are code smells that have the most impact on quality attributes and bug propensity [20], while also being considered as most problematic by developers [31].

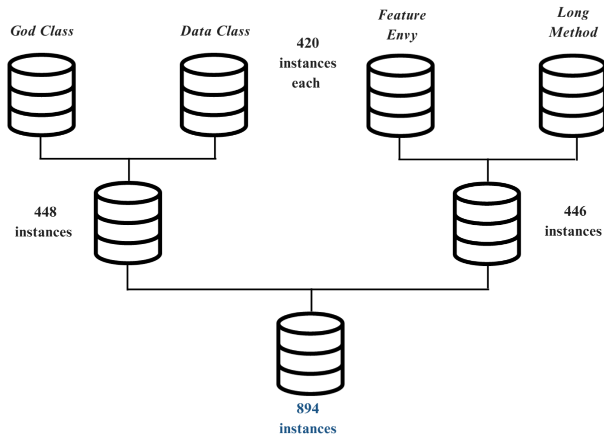

Figure 1: Overview of the dataset combination process

After the algorithm execution is completed, both data subsets, at class (448 instances) and method (446 instances) levels, are combined to form the final dataset with 894 instances. The algorithm generates, at most, $((3 * Ncodesmelltypes) + 1)$ different types of code smell severity classes, one of them being the no-smell class. In our specific case, 13 classes are generated, as the number of code smells is 4, i.e., GC, DC, FE, LM. Table 3 presents the details of the generated classes, their number of instances, and percentages of distribution by severity.

Algorithm 1 Dataset Combination Algorithm

```
1:  while there are instances to be analyzed do
2:    Search for duplicate instances (at class or method level)
3:    if duplicate instances found then
4:      if severities of the cases are different then
5:        Exclude the lowest severity instance
6:        if severity is different from no smell then
7:          Change the class name of the remaining instance to
            "severity + code smell"
8:        end if
9:      else if severity is different from no smell then
10:       Exclude the FE or DC instance
11:       Change the class name of the remaining instance to
          "severity + GC or LM"
12:     else
13:       Exclude either of the two instances
14:     end if
15:   else if severity is different from no smell then
16:     Change the class name of the instance to "severity + code
        smell"
17:   end if
18: end while
```

Table 3: Details of the generated classes

Code Smell	Severity ID	Class Name	# Instances	% Severity
0	0	No code smell	413	-
1	1	Non-severe Feature-envy	13	2.70
1	2	Feature-envy	39	8.11
1	3	Severe Feature-Envy	11	2.29
1	4	Non-severe Long-method	8	1.66
1	5	Long-method	90	18.71
1	6	Severe Long-method	34	7.07
1	7	Non-severe Data-class	32	6.65
1	8	Data-class	77	16.01
1	9	Severe Data-class	37	7.69
1	10	Non-severe God-class	9	1.87
1	11	God-class	33	6.86
1	12	Severe God-class	98	20.37

It is important to highlight that negative instances, without code smell, constitute 46.20% of the combined dataset, while positive instances, with code smells, total 53.80%.

Because of the complexity of this strategy, the code smell severity classifying can become more challenging, requiring a solution that goes beyond the simple use of ML techniques, such as optimization of machine hyperparameters, data standardization, and feature selection techniques.

4 ENSEMBLES FOR DETECTION AND SEVERITY CLASSIFICATION

Ensemble methods play an important role in improving the performance of machine learning models for code smell detection and

have been suggested by several authors [6, 35]. The core idea behind the ensemble methods is to combine multiple classifiers to obtain superior results compared to a standalone best classifier. This approach is well presented and tutorials have been provided to guide specific practitioners in building set-based classification systems [36, 38].

The ensembles can be categorized as homogeneous or heterogeneous. Homogeneous ensembles are built with classifiers of the same type, trained on different views of the dataset. On the other hand, heterogeneous ensembles combine different types of classifiers [2]. Homogeneous ensemble methods, particularly the Random Forest (RF) technique, have become widely prevalent among novel techniques for detecting code smells [5, 12, 13, 19, 35].

Another ensemble method that obtained good results in code smell detection was Category Boosting (CatBoost) [3], which was used to detect the God-class, obtaining the highest performance among 27 other classifiers used, in a real scenario where data is generally unbalanced due to the inherent nature of code smells. CatBoost employs the Sort Boosting algorithm like-replacement for the traditional gradient estimation method. This substitution reduces the bias in the gradient estimate, ultimately improving the generalization ability of the model [46].

In the case of code smell severity classification, a multiclass problem, Extreme Gradient Boosting (XGBoost) stands out for its prominent results [1, 11, 18]. It exhibits robust performance, even when the dataset lacks comprehensive preprocessing, and is an important tree-based machine-learning algorithm known for its superior performance and speed. Initially developed by Tianqi Chen and overseen by the Distributed Machine-Learning Community (DMLC) organization, it is recognized for its effectiveness in handling structured and tabular data. Given its simplicity and effectiveness, XGBoost has gained widespread popularity, especially for code smell detection [11].

Unlike most approaches that rely on RF to detect code smells, in our approach, we combine the excellent accuracy performance of the CatBoost model in the context of data imbalance [3] to detect positive and negative instances of code smell severity. We then use the XGBoost model in a multi-target class scenario [11] to capture the nuances of each class, focusing solely on the positive instances recovered from the previous step, providing a better classification for their severities.

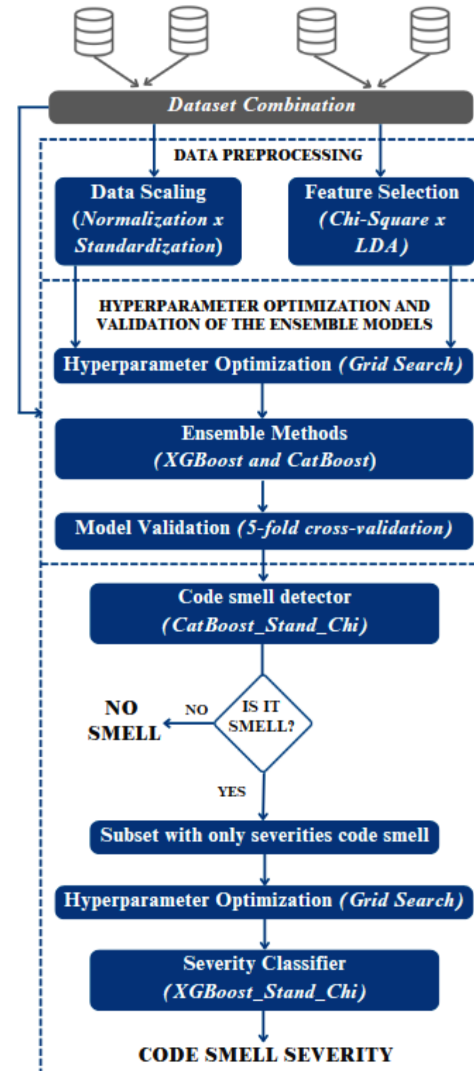

Figure 2: Overview of the proposed approach

5 EXPERIMENTS AND RESULTS

This section details the experiments on the combined dataset, applying data preprocessing techniques (data scaling and feature selection), and ensemble methods for detecting and classifying the severity of code smells.

5.1 Experiments Setup

Figure 2 illustrates the proposed approach overview for detecting code smells and classifying their severity, under which the experiments were carried out.

For these experiments, we used a dataset combined from the following datasets, Arcelli Fontana and Zanoni [4], Nanda and Chhabra [29], going through the procedure described in section 3 resulting in a dataset with 894 instances, covering 4 code smell types, i.e., GC, DC, FE, and LM. Furthermore, the dataset was split into 70% for training and hyperparameter optimization and 30% for testing, and model validation used a 5-fold cross-validation setup on the entire dataset, with or without preprocessing. This aims to ensure that models have a sufficient amount of data to learn from, a robust set to test, and enough instances of each class to validate while maintaining the relevance of results.

5.1.1 Data Preprocessing. Data Preprocessing always represents a crucial aspect in building more accurate ML models, since it covers the cleaning and transformation of raw data, which facilitates the analysis and identification of the most relevant information, thus contributing to improving the model's performance [33]. Before applying other preprocessing techniques to a combination dataset, which includes labels and metrics as features, it is crucial to address specific issues within that set, such as the presence of duplicate data and missing values. Moreover, the exploratory analysis of the data

set demonstrated that the range of values between the attributes is different, and the difference between each attribute's minimum and maximum values (e.g., LOC_method and LOC_package) can vary at high values. Furthermore, some dataset attributes have several outliers, such as ATFD_method and AMWNAMM_type. These hidden characteristics can bias the results of ML models.

The process was, then, initiated by eliminating code smell instances duplicated (section 3), which are always related to the same level and never between different levels. Next, imputations were performed for the missing values of 21 attributes, as these attributes are specific to method code smells (84 attributes), and do not contain values in class datasets (63 attributes). This imputation was necessary so that ML models could act appropriately, without harming the classifications of class datasets.

Regarding missing values, i.e., method metrics (NOP, CC, ATFD, FDP, CM, MAXNESTING, LOC, CYCLO, NMCS, NOLV, MaMCL, NOAV, LAA, FANOUT, CFNAMM, ATLD, CLNAMM, CINT, MeMCL, CDISP) in the GC and DC instances, imputation was performed using the mode value because the sample values for each attribute do not follow a normal distribution. This value was calculated considering only the no code smell results in instances of the FE and LM classes. Furthermore, no "is_Static_method" attribute was assigned a value of zero in each instance. The underlying hypothesis is that machine interpretation learning models are not instances of FE or LM. With the other independent values related to GC and DC, the models will be able to classify them. Finally, the nominal categorical variables were transformed into ordinal categorical variables, i.e., "modifier_type" and "visibility_type".

Once this preprocessing phase was completed, Normalization and Standardization techniques were applied to achieve resource scaling. Finally, to select the most impactful features in the combined dataset, improving the performance of the models by focusing on software metrics that play a significant role in distinguishing between similar functions in design patterns, were applied and compared as follows techniques: LDA and Chi-Square. LDA is a supervised machine learning algorithm specialized in identifying distinctive characteristics between different groups or classes [34], while the chi-square-based feature selection technique is employed on categorical attributes of a dataset [11].

In the case of Chi-Square, we used the selected features by Dewangan et al. [12] (see Table 4). These features represent some quality dimensions in Object-Oriented (OO) software engineering, for example, size, complexity, coupling, encapsulation, and cohesion, which are calculated at the level of method, class, or both. The difference between our study and the reference study [12] is that, unlike their study, which uses these features on separate datasets, we selected the used features by them at least once in each data set, applying them all together to our combined dataset.

5.1.2 Hyperparameter Optimization and Validation of the Ensemble Models. After applying data preprocessing techniques (data scaling and feature selection), we proceeded with the optimization of the hyperparameters, using the Grid Search algorithm, to determine the most appropriate settings for the hyperparameters of each ensemble method. These hyperparameters were configured to optimize the model based on the f1-macro metric, thus without considering the imbalance between the target classes. Also, to improve the models' performance, we used a 5-fold cross-validation method in the hyperparameter optimization technique. This approach was selected due to the presence of classes with extremely limited instances.

Next, the accuracy results of the ensemble models combined or not with data preprocessing techniques were analyzed to choose the most appropriate model for each phase of the proposed approach, i.e., a code smell detector and a code smell severity classifier.

Lastly, this study has used a validation methodology to calculate each experiment's performance to avoid overfitting the algorithm on the test dataset. Ensembles are calculated using 5-fold cross-validation to divide the datasets into five segments, five times for the algorithm training. In each replication dataset, one part is evaluated as the test set, and the others are evaluated for the training set.

5.1.3 Code Smell Detector and Severity Classifier. The most appropriate model was chosen for each stage of our approach, after configuring hyperparameters, validating the models, and analyzing the performance improvements of each ensemble method using data scaling and feature selection techniques.

Unlike Arcelli Fontana and Zanoni [4] or Nanda and Chhabra [29], we use ensemble methods with data scaling and feature selection to detect and classify code smells. The former used normalization without showing its use and did not use sophisticated feature selection approaches while the latter did not use data scaling and feature selection, basing their approach solely on the ensemble method and data balancing.

In our approach, the first step consists of a code smell detection module composed of the CatBoost ensemble method, standardization, and Chi-square. Allowing, thus, the detection of instances with or without code smell following a binary approach, which is less complex than the next, and reducing the model cost for detecting negative instances.

The following step aims to classify the severity of code smells, using standardization and chi-square, but with the XGBoost ensemble method in a multiclass approach. However, this severity classification is not restrained by the large number of negative instances, common in this problem type, as these instances are previously discarded before this second step of the approach, leaving only positive instances to be dealt with.

In the end, the set of experiments was converted, and to evaluate the quality of the ensemble method model, four performance parameters were considered: Precision (P), Recall (R), F1-Score (F1), and Accuracy. These parameters are calculated using a confusion matrix that contains the actual and predicted information estimated by the design pattern detection classifications [9], i.e., True Positive (TP), False Positive (FP), True Negative (TN), and False Negative (FN).

5.2 Results

The ensemble model experiments were conducted in three ways on the combined dataset: without data preprocessing; with only one of the data preprocessing techniques; and with a combination of data scaling and feature selection techniques. This determined the best combination of data scaling, feature selection, and machine learning techniques for code smell detection and severity classification on the combined dataset for each approach stage and their impacts on the models. First, the accuracy of the ML models on

Table 4: Selected Features by Dewangan et al. [12] with Chi-Square

Quality dimension	Metric label and citation	Metric name	Granularity	Definition
Size	AMW [10, 44]	Average Methods Weight	Class	The sum of complexity of the methods that are defined in the class
	CYCLO [26, 44]	Cyclomatic Complexity	Method	The maximum number of linearly independent paths in a method. A path is linear if there is no branch in the execution flow of the corresponding code
	LOC [22]	Lines of Code	Class, Method	The number of lines of code of an operation or a class, including blank lines and comments
	LOCNAMM [16]	Lines of Code Without Accessor or Mutator Methods	Class	The number of lines of code of a class, including blank lines and comments and excluding accessor and mutator methods and corresponding comments
	NOA	Number of Attributes	Class	Number of attributes of a class.
	NOMNAMM [16]	Number of Not Accessor or Mutator Methods	Class	The number of methods defined locally in a class, counting public and private methods, excluding accessor or mutator methods
	WMC [10, 44]	Weighted Methods Count	Class	The sum of complexity of the methods that are defined in the class
	WMCNAMM [16]	Weighted Methods Count of Not Accessor or Mutator Methods	Class	The sum of complexity of the methods defined in the class, but are not accessor or mutator methods
	WOC [25]	Weight of Class	Class	The number of "functional" public methods divided by the total number of public members
Complexity	ATFD [25]	Access to Foreign Data	Class, Method	The number of attributes from unrelated classes belonging to the system, accessed directly or by invoking accessor methods
	ATLD [16]	Access to Local Data	Method	The number of attributes from the current classes accessed by the measured method directly or by invoking accessor methods
	FANOUT [27]	-	Class, Method	Number of called classes
	FDP [27]	Foreign Data Providers	Method	The number of classes in which the attributes accessed - in conformity with the ATFD metric - are defined
	MAXNETing [27]	Maximum Nesting Level	Method	The maximum nesting level of control structures within an operation
	NOAV [27]	Number of Accessed Variables	Method	The total number of variables accessed directly or through accessor methods from the measured operation
	NOLV [25]	Number of Local Variable	Method	The total number of local variables accessed directly or through accessor methods from the measured operation
	RFC [10, 44]	Response for a Class	Class	The size of the response set of a class
Coupling	CFNAMM [16]	Called Foreign Not Accessor or Mutator Methods	Class, Method	The number of called not accessor or mutator methods declared in unrelated classes concerning: I) the one that declares the measured for each method; and ii) the measured one, for class
	CINT [25]	Coupling Intensity	Method	The number of distinct operations called by the measured operation
Encapsulation	DIT [10, 44]	Depth of Inheritance Tree	Class	The number of ancestor classes measures the maximum length from the class node to the tree's root
Cohesion	TCC [7, 8, 44]	Tight Class Cohesion	Classe	The normalized ratio between the number of methods directly connected with other methods through an instance variable and the total number of possible connections between methods

the combined dataset without data preprocessing and with data scaling techniques were compared. Next, ensemble methods performed well without preprocessing; normalization harmed models, while standardization improved validation with less standard deviation, as standardization handled outliers better [15]. Then, feature selection combined with data scaling techniques were applied to improve the accuracy of models, simplifying them and making the results obtained more understandable, mainly with the chi-square technique. The GitHub repository [4] links article results to available files. Refer to README.md for details.

[4]https://github.com/fabiorosario/Code-smell-severity-classification

The results of these experiments revealed that standardization achieved its highest accuracy in the CatBoost model, providing 80.67%, even without data processing (see Table 5).

Table 5: Accuracies with Normalization and Standardization

Ensemble Method	No data scaling	Normalization	Standardization
RF	**79.55**	75.09	**79.55**
XGBoost	**79.18**	76.21	**79.18**
CatBoost	**80.67**	76.95	**80.67**

However, this result did not surpass the effectiveness of combining data standardization, chi-square, and ensemble methods, which achieved 81.04% and 81.41%, in the XGBoost and CatBoost models, respectively (see Table 6).

Table 6: Accuracies with LDA and Chi-square

Ensemble Method	LDA	Chi-square	Chi-Square Standardization
RF	74.72	**78.07**	**78.07**
XGBoost	76.58	80.30	**81.04**
CatBoost	75.84	79.55	**81.41**

Given this, our approach for detecting and classifying code smell severity included two steps. The first step aims to detect code smells from instances with severity, using the Standardization data preprocessing technique, Chi-Square for feature selection, and the CatBoost ensemble method to binary identify the presence or absence of code smell. The following step removes the negative instances and classifies the code smell severity of the positive instances, using the standardization, chi-square, and XGBoost. This approach obtained 85% accuracy, as described in Table 7.

Table 7: Values for the achieved results (%)

Severity	P	R	F1	Support
0-No code smell	**98**	88	93	129
1-Non-severe Feature-envy	**100**	50	67	6
2-Feature-envy	54	**58**	56	12
3-Severe Feature-Envy	**100**	**100**	**100**	1
4-Non-severe Long-method	0	0	0	2
5-Long-method	**79**	76	77	29
6-Severe Long-method	64	**100**	78	7
7-Non-severe Data-class	**75**	**75**	**75**	8
8-Data-class	**83**	76	79	25
9-Severe Data-class	71	**83**	77	12
10-Non-severe God-class	0	0	0	0
11-God-class	**62**	50	56	10
12-Severe God-class	83	**89**	86	28
Accuracy		**85**		

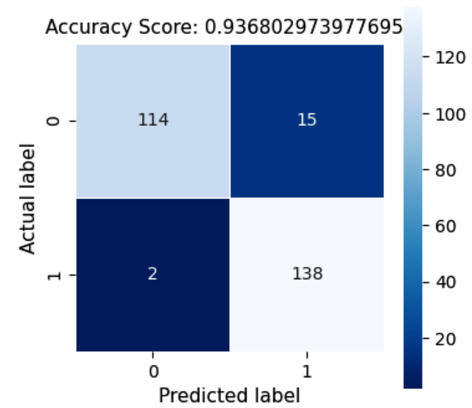

Figure 3: Confusion Matrix of the Detection Phase

	Non-severe FE	FE	Severe FE	Non-severe LM	LM	Severe LM	Non-severe DC	DC	Severe DC	Non-severe GC	GC	Severe GC
Non-severe FE	3	3	0	0	0	0	0	0	0	0	0	0
FE	0	7	0	0	4	1	0	0	0	0	0	0
Severe FE	0	0	1	0	0	0	0	0	0	0	0	0
Non-severe LM	0	0	0	0	2	0	0	0	0	0	0	0
LM	0	3	0	1	22	3	0	0	0	0	0	0
Severe LM	0	0	0	0	0	7	0	0	0	0	0	0
Non-severe DC	0	0	0	0	0	0	6	2	0	0	0	0
DC	0	0	0	0	0	0	2	19	4	0	0	0
Severe DC	0	0	0	0	0	0	0	2	10	0	0	0
Non-severe GC	0	0	0	0	0	0	0	0	0	0	0	0
GC	0	0	0	0	0	0	0	0	0	0	5	5
Severe GC	0	0	0	0	0	0	0	0	0	0	3	25

Figure 4: Confusion Matrix of the Severity Classification Phase

The confusion matrix presented in Figure 3 suggests that the approach's code smell detector can detect negative instances reasonably well, achieving measures of precision, recall, f1-score, and accuracy in the order of 98%, 88%, 93%, and 94%, respectively.

Figure 4 shows the confusion matrix, in which the hit-and-miss frequencies achieved by the severity classifier for each class are shown.

The worst-case scenario happens when the model indicates a severity that is less than what it is, as it can significantly affect the quality and life cycle of the software due to failure to maintain a more serious severity. In this sense, our model behaves well, as this happened only a few times:

- 1 instance was classified as Non-severe LM when it should have been classified as LM, that is 3% of 29 instances.
- 2 instances were classified as Non-severe DC when they should have been classified as DC, 8% of 25 instances.
- 2 instances were classified as DC when they should have been classified as Severe DC, 17% of 12 instances.
- 3 instances were classified as GC when they should have been classified as Severe GC, 11% of 28 instances.

Another scenario that can result in unnecessary maintenance costs occurs when a less severe code smell is prioritized over a more serious one due to incorrect classification. These also happened a few times as in:

- 1 instance was classified as Severe LM when they should have been classified as FE, 8% of 12 instances.
- 3 instances were classified as Severe LM when they should have been classified as LM, 10% of 29 instances.

- 2 instances were classified as DC when they should have been classified as Non-severe DC, 25% of 8 instances.
- 4 instances were classified as Severe DC when they should have been classified as DC, 16% of 25 instances.
- 5 instances were classified as Severe GC when they should have been classified as GC, 50% of 10 instances.

Another important information that can be extracted from the confusion matrix is that our model managed to predict all instances of the DC and GC classes in one of their three severity levels, even with interference from other instances at the method level, i.e., FE and L.M. However, the same fact was not observed in the severities of the FE and LM classes, where in 26% of cases the model predicted the LM classes, when it should have predicted FE, in 5 of the 19 cases. Also, in 8% of the cases, the model predicted the FE classes, when it should have predicted LM, in 3 of the 38 cases. This indicates that handling multi-label classes may not be sufficient to deal with method-level interference in the data combination process (section 3) and suggests more specialized handling for these instances. It is also important to realize that our approach got all instances right when considering each level, whether method or class, that is, instances at the class level did not interfere with the classification of instances at the method level and vice versa. Even with the simple imputation of missing data in class instances, the model exhibited this expected behavior.

5.3 Comparison with results from relevant research studies

Here, we elaborate a comparative summary of the results of the proposed approach to other relevant research studies. To the best of our knowledge and based on the available literature, we have identified only six works [1, 4, 11, 18, 29, 37] addressing the detection of the severity of code smells, with the first five not providing measured values for precision, recall, or f1-score. Dewangan et al. [11] was the only one that raised these questions, and, given this scenario, we carried out two comparisons. The first, using the mean accuracy of each study with our approach, considering that our dataset is unique for all types of code smells, and the second, comparing our results with those of Dewangan et al. [11], who reported all metrics.

In the first comparison (Table 8), our approach presented superior performance only to the study of Arcelli Fontana and Zanoni [4], recording an average accuracy of 85% compared to 84%.

Table 8: Average accuracy assessment

APPROACH	DC	GC	FE	LM	AVG
Arcelli Fontana and Zanoni [4]	74	77	92	93	84.00
Nanda and Chhabra [29]	**97**	**98**	**98**	99	**98.00**
Abdou and Darwish [1]	93	92	97	97	94.75
Rao et al. [37]	83	85	96	99	90.75
Hu et al. [18]	79	78	92	92	85.25
Dewangan et al. [11]	88.22	86	96	**99.12**	92.34
Our approach	Single Dataset				85.00

Concerning the other works, we experienced a difference that varied from 0.25% to 13%, unfavorable to our approach. The following works can be highlighted, since Nanda and Chhabra [29] applied the SMOTE resampling technique and achieved a remarkable average accuracy of 98%, and the 99.12% accuracy recorded by the XGBoost approach of Dewangan et al. [11].

In the comparison provided by Table 9, our precision, recall, and f1-score results were below the results obtained by Dewangan et al. [11], with the smallest differences being found in the DC dataset, 6% in all metrics. However, the biggest differences were recorded in LM precision (20%), FE recall (22%), and FE and LM F1-score (17%).

Table 9: Precision, Recall and F1-Score Evaluation

Code Smell	Metrics	Dewangan et al. [11]	Our Approach
DC	P (%)	88	*82*
	R (%)	87	*81*
	F1 (%)	87	*81*
GC	P (%)	90	81
	R (%)	90	76
	F1 (%)	90	78
FE	P (%)	97	88
	R (%)	**96**	**74**
	F1 (%)	96	79
LM	P (%)	**100**	**80**
	R (%)	100	88
	F1 (%)	**100**	**83**

In conclusion, this section provides a comprehensive analysis of the results obtained by the proposed approach in comparison with other relevant research studies in detecting the severity of code smells. The comparison revealed that, although outperforming Arcelli Fontana and Zanoni [4], our approach showed unfavorable differences with other works. This evaluation highlights the continued importance of improving approaches and techniques in detecting code smells to advance the effectiveness and reliability of these models in real-world scenarios.

6 THREATS TO VALIDITY

Various threats to validity may influence the integrity and accuracy of the results and conclusions of this research. This section summarizes some implications for our ML-based code smell detection and classification approach.

The construct validity of our study is mainly linked to the merging of the dataset. We selected well-known datasets: i) the FE and LM dataset from Arcelli Fontana and Zanoni [4], which was used in all related works, and the GC and DC dataset from Nanda and Chhabra [29], which corrected some inaccuracies in the original dataset [4]. To mitigate this threat, we remove duplicate instances and perform missing value imputations so that the ML models can act appropriately. Most importantly, we present and follow a data combination algorithm that allows the process to be verified and repeated.

Regarding external validity, our combined dataset is a more realistic representation of real-world code smells at both the method and class levels. By addressing code smells at both levels and ensuring that the dataset reflects a variety of programming scenarios, we increase the external validity of our findings. This means that our results are more likely to be generalizable to other software projects

and programming environments beyond our specific dataset, including those written in different programming languages, than previous work.

Therefore, the changes in data and models contributed to reducing classification bias both by not having duplicate instances and by standardizing attribute scales to deal with outliers. It is worth highlighting that mode as a strategy for filling in missing data was important in the results of the models, mainly because no classification errors were identified in a class instance being predicted as a method instance or vice versa.

Another important factor is the reliability of the conclusion, which refers to the threats that influence the ability to draw the same conclusions if the procedure is repeated under the same conditions. The main issue related to this category was the configuration of the experiments performed on the combined dataset. To mitigate this threat, this work analyzes, details, and makes publicly available all stages of the data combination process, hyperparameter optimization (grid search), model validation (5-fold crossover), comparison, and choice of techniques employed (data scaling, feature selection, and ML).

7 CONCLUSION

Code smell detection is a data-driven approach that aims to ensure code quality by identifying if a code segment breaches essential design principles, which can degrade design quality. Nevertheless, previous research studies primarily focus on categorizing the severity of individual code smell types within specific datasets, diverging from real-world software scenarios because this overlooks how different code smell types interact.

To help smaller teams or those with tight deadlines reduce costs and prioritize issues, the objective of this study is to detect and classify the severity of code smell in a single dataset covering instances of four distinct types of code smell, i.e., God-class, Data-Class, Feature-Envy, and Long-Method. In contrast to previous studies that focus on this task by analyzing datasets with only one specific type of code smell, our approach comes closer to the real scenario of existing software, considering the influence different types of code smells exert on each other.

In this context, our research work develops a model to detect and classify the severity of code smells through the combined use of scaling datasets, Normalization and Standardization techniques, and the selection of features with the LDA and Chi-square techniques. Furthermore, we apply two ensemble methods, i.e., XGBoost and CatBoost, to detect and classify code smell severity. We also use the Grid Search algorithm for hyperparameter optimization and 5-fold cross-validation in the best machine learning models.

The results indicate that Standardization reached its highest accuracy in the CatBoost model, recording 80.67%. In contrast, Normalization dropped the accuracy, with CatBoost having the biggest drop (3,72%). LDA showed worse values compared to the combined dataset with all features, with the CatBoost method dropping almost 6% in its performance. However, the achieved results by the combined dataset without feature selection technique did not surpass the effectiveness of the combination of dataset standardization techniques, ensemble methods, and Chi-square, which achieved values of 81.04% and 81.41% in the XGBoost and CatBoost models, respectively.

Despite encountering interference from other instances at the method level, specifically FE and L.M., our model successfully predicted all instances of the DC and GC classes in one of their three severity levels. Notably, our approach demonstrated accuracy across all instances when considering each level—whether method or class. This implies that instances at the class level did not disrupt the classification of instances at the method level, and vice versa.

However, when compared with other relevant works in this area of research, our approach performed better than the study of Arcelli Fontana and Zanoni [4], recording an average accuracy of 85% compared to the 84% of the aforementioned author. Regarding precision, recall, and f1-score, our results are outperformed by those obtained by Dewangan et al. [11], with the smallest differences being discovered in the DC dataset, with 6% across all as measurements. The biggest differences, however, were recorded in LM precision (20%), FE recall (22%), and LM f1-score (17%).

The main findings, in addition to those already mentioned about normalization, standardization, and feature selection, are: i) The multiclass approach, despite not being influenced by the large number of negative instances, lacks the treatment of unbalanced data to the most superficial severities (minority) in comparison to the most serious ones (majority), e.g., using SMOTE at this stage to create other relevant points of minority classes, without compromising the quality of the sample; ii) Although ensemble methods play an important role in this area, as long as their hyperparameters are adequately optimized, with the help of appropriate techniques, they were not sufficient to solve the problem of code smell severities class imbalances, even with data scaling and feature selection.

Given these results, in future work, we will focus on carrying out a more detailed analysis of each independent attribute of the dataset and verifying, according to the individual distribution of each one, the imputation of missing values in a more appropriate way. Another possibility for future work will be to deal with multi-label classes with specific techniques for this type of problem, since, in our work, these instances were used to balance the classes corresponding to their labels in comparison with other classes of the dataset. This opens up the possibility of using a more sophisticated technique for balancing minority classes, such as the SMOTE technique, which was successful in Nanda and Chhabra [29].

Finally, analysis of the results obtained by the proposed approach in comparison with other relevant research studies in detecting the severity of code smells suggests the need to improve approaches and techniques in detecting code smells to advance the effectiveness and reliability of these models in real-world scenarios. In that regard, our ML severity classification approach aims to create an ML model that can, e.g., be transferred to other programming languages (although this is not the focus of this paper). With this most robust approach and better generalization of the models, it is possible to overcome the limitations of available datasets, which often have imbalanced samples, inadequate severity level support, and are primarily Java-based [39, 47], despite C#'s prominence in code smell discussions [40], most annotated datasets are Java-based [47]. Despite this, the combination process and the ML-trained model, as devised, may be applied to code smell instances written in other programming languages.

REFERENCES

[1] Ashraf Abdou and Nagy Darwish. 2024. Severity classification of software code smells using machine learning techniques: A comparative study. *Journal of Software: Evolution and Process* 36, 1 (2024), e2454. https://doi.org/10.1002/smr.2454 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2454

[2] Amal Alazba and Hamoud Aljamaan. 2021. Code smell detection using feature selection and stacking ensemble: An empirical investigation. *Information and Software Technology* 138 (2021), 106648. https://doi.org/10.1016/j.infsof.2021.106648

[3] Khalid Alkharabsheh, Sadi Alawadi, Victor R. Kebande, Yania Crespo, Manuel Fernández-Delgado, and José A. Taboada. 2022. A comparison of machine learning algorithms on design smell detection using balanced and imbalanced dataset: A study of God class. *Information and Software Technology* 143 (2022), 106736. https://doi.org/10.1016/j.infsof.2021.106736

[4] Francesca Arcelli Fontana and Marco Zanoni. 2017. Code smell severity classification using machine learning techniques. *Knowledge-Based Systems* 128 (2017), 43–58. https://doi.org/10.1016/j.knosys.2017.04.014

[5] Francesca Arcelli Fontana, Mika V. Mäntylä, Marco Zanoni, and Alessandro Marino. 2016. Comparing and Experimenting Machine Learning Techniques for Code Smell Detection. *Empirical Softw. Engg.* 21, 3 (jun 2016), 1143–1191. https://doi.org/10.1007/s10664-015-9378-4

[6] Muhammad Ilyas Azeem, Fabio Palomba, Lin Shi, and Qing Wang. 2019. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology* 108 (2019), 115–138. https://doi.org/10.1016/j.infsof.2018.12.009

[7] L.C. Briand, J.W. Daly, and J.K. Wust. 1999. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering* 25, 1 (1999), 91–121. https://doi.org/10.1109/32.748920

[8] William H. Brown, Raphael C. Malveau, Hays W. "Skip" McCormick, and Thomas J. Mowbray. 1998. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis* (1st ed.). John Wiley & Sons, Inc., USA.

[9] Cagatay Catal. 2012. Performance Evaluation Metrics for Software Fault Prediction Studies. *Acta Polytechnica Hungarica* 9 (01 2012).

[10] O. Ciupke. 1999. Automatic detection of design problems in object-oriented reengineering. In *Proceedings of Technology of Object-Oriented Languages and Systems - TOOLS 30 (Cat. No.PR00278)*. IEEE, Santa Barbara, CA, USA, 18–32. https://doi.org/10.1109/TOOLS.1999.787532

[11] Seema Dewangan, Rajwant Singh Rao, Sripriya Roy Chowdhuri, and Manjari Gupta. 2023. Severity Classification of Code Smells Using Machine-Learning Methods. *SN Computer Science* 4, 5 (2023). https://doi.org/10.1007/s42979-023-01979-8

[12] Seema Dewangan, Rajwant Singh Rao, Alok Mishra, and Manjari Gupta. 2021. A novel approach for code smell detection: An empirical study. *IEEE Access* 9 (2021), 162869–162883. https://doi.org/10.1109/ACCESS.2021.3133810

[13] Seema Dewangan, Rajwant Singh Rao, Alok Mishra, and Manjari Gupta. 2022. Code Smell Detection Using Ensemble Machine Learning Algorithms. *Applied Sciences (Switzerland)* 12, 20 (2022), 10321. https://doi.org/10.3390/app122010321

[14] Dario Di Nucci, Fabio Palomba, Damian A. Tamburri, Alexander Serebrenik, and Andrea De Lucia. 2018. Detecting code smells using machine learning techniques: Are we there yet?, In IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). *25th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2018 - Proceedings* 2018-March (2018), 612–621. https://doi.org/10.1109/SANER.2018.8330266

[15] Katti Faceli, Ana Carolina Lorena, João Gama, Tiago Agostinho de Almeida, and André Carlos Ponce de Leon Ferreira de Carvalho. 2021. *Inteligência artificial: uma abordagem de aprendizado de máquina*. LTC, Brazil.

[16] Vincenzo Ferme. 2013. *JCodeOdor: A Software Quality Advisor Through Design Flaws Detection.* Ph. D. Dissertation. Università degli Studi di Milano-Bicocca.

[17] Francesca Arcelli Fontana, Vincenzo Ferme, Marco Zanoni, and Riccardo Roveda. 2015. Towards a prioritization of code debt: A code smell Intensity Index. In *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)* (7 ed.). IEEE, Bremen, Germany, 16–24. https://doi.org/10.1109/MTD.2015.7332620

[18] Wenhua Hu, Lei Liu, Peixin Yang, Kuan Zou, Jiajun Li, Guancheng Lin, and Jianwen Xiang. 2023. Revisiting "code smell severity classification using machine learning techniques", In IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC), Shahriar H., Teranishi Y., Cuzzocrea A., Sharmin M., Towey D., Majumder AKM.J.A., Kashiwazaki H., Yang J.-J., Takemoto M., Sakib N., Banno R., and Ahamed S.I. (Eds.). *Proceedings - International Computer Software and Applications Conference* 2023-June (2023), 840–849. https://doi.org/10.1109/COMPSAC57700.2023.00113

[19] Aleksandar Kovačević, Jelena Slivka, Dragan Vidaković, Katarina-Glorija Grujić, Nikola Luburić, Simona Prokić, and Goran Sladić. 2022. Automatic detection of Long Method and God Class code smells through neural source code embeddings. *Expert Systems with Applications* 204 (2022), 117607. https://doi.org/10.1016/j.eswa.2022.117607

[20] Guilherme Lacerda, Fabio Petrillo, Marcelo Pimenta, and Yann Gaël Guéhéneuc. 2020. Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software* 167 (2020), 110610. https://doi.org/10.1016/j.jss.2020.110610

[21] Michele Lanza and Radu Marinescu. 2007. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, Springer-Verlag Berlin Heidelberg 2006.

[22] Mark Lorenz and Jeff Kidd. 1994. *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc., USA.

[23] Lech Madeyski and Tomasz Lewowski. 2020. MLCQ: Industry-Relevant Code Smell Data Set. In *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering* (24 ed.) (Trondheim, Norway) *(EASE '20, 24)*. Association for Computing Machinery, New York, NY, USA, 342–347. https://doi.org/10.1145/3383219.3383264 03/04/2024.

[24] Cristina Marinescu, Radu Marinescu, Petru Mihancea, Daniel Ratiu, and Richard Wettel. 2005. iPlasma: An Integrated Platform for Quality Assessment of Object-Oriented Design., In Proceedings of the 21st IEEE International Conference on Software Maintenance - Industrial and Tool volume, ICSM 2005, 25-30 September 2005, Budapest, Hungary. *Proceedings of ICSM* 1, 14, 77–80.

[25] R. Marinescu. 2005. Measurement and quality in object-oriented design. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE, Budapest, Hungary, 701–704. https://doi.org/10.1109/ICSM.2005.63

[26] T.J. McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* SE-2, 4 (1976), 308–320. https://doi.org/10.1109/TSE.1976.233837

[27] Radu Marinescu Michele Lanza. 2006. *Object-Oriented Metrics in Practice*. Springer, Berlin, Heidelberg. XIV, 207 pages. https://doi.org/10.1007/3-540-39538-5

[28] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. 2010. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transactions on Software Engineering* 36, 1 (2010), 20–36. https://doi.org/10.1109/TSE.2009.50

[29] Jatin Nanda and Jitender Kumar Chhabra. 2022. SSHM: SMOTE-stacked hybrid model for improving severity classification of code smell. *International Journal of Information Technology (Singapore)* 14, 5 (2022), 2701–2707. https://doi.org/10.1007/s41870-022-00943-8

[30] Kwankamol Nongpong. 2012. *Integrating "code smells" detection with refactoring tool support.* Ph. D. Dissertation. University of Wisconsin at Milwaukee, USA. Advisor(s) Boyland, John Tang. AAI3523928.

[31] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. 2014. Do They Really Smell Bad? A Study on Developers' Perception of Bad Code Smells. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, Victoria, BC, Canada, 101–110. https://doi.org/10.1109/ICSME.2014.32

[32] Fabio Palomba, Dario Di Nucci, Michele Tufano, Gabriele Bavota, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. 2015. Landfill: An Open Dataset of Code Smells with Public Evaluation. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories* (12 ed.). IEEE, Florence, Italy, 482–485. https://doi.org/10.1109/MSR.2015.69

[33] Anubha Parashar, Apoorva Parashar, Weiping Ding, Mohammad Shabaz, and Imad Rida. 2023. Data preprocessing and feature selection techniques in gait recognition: A comparative study of machine learning and deep learning approaches. *Pattern Recognition Letters* 172 (2023), 65–73. https://doi.org/10.1016/j.patrec.2023.05.021

[34] Archana Patnaik and Neelamadhab Padhy. 2022. Does Code Complexity Affect the Quality of Real-Time Projects? Detection of Code Smell on Software Projects Using Machine Learning Algorithms. In *Proceedings of the International Conference on Data Science, Machine Learning and Artificial Intelligence* (<conf-loc>, <city>Windhoek</city>, <country>Namibia</country>, </conf-loc>) *(DSMLAI '21')*. Association for Computing Machinery, New York, NY, USA, 178–185. https://doi.org/10.1145/3484824.3484911

[35] Fabiano Pecorelli, Fabio Palomba, Dario Di Nucci, and Andrea De Lucia. 2019. Comparing heuristic and machine learning approaches for metric-based code smell detection, In IEEE/ACM 27th International Conference on Program Comprehension (ICPC). *IEEE International Conference on Program Comprehension* 2019-May (2019), 93–104. https://doi.org/10.1109/ICPC.2019.00023

[36] R. Polikar. 2006. Ensemble based systems in decision making. *IEEE Circuits and Systems Magazine* 6, 3 (2006), 21–45. https://doi.org/10.1109/MCAS.2006.1688199

[37] Rajwant Singh Rao, Seema Dewangan, Alok Mishra, and Manjari Gupta. 2023. A study of dealing class imbalance problem with machine learning methods for code smell severity detection using PCA-based feature selection technique. *Scientific Reports* 13, 1 (2023). https://doi.org/10.1038/s41598-023-43380-8

[38] Lior Rokach. 2010. Ensemble-based classifiers. *Artificial Intelligence Review* 33 (2010), 1–39.

[39] Jelena Slivka, Nikola Luburić, Simona Prokić, Katarina-Glorija Grujić, Aleksandar Kovačević, Goran Sladić, and Dragan Vidaković. 2023. Towards a systematic approach to manual annotation of code smells. *Science of Computer Programming* 230 (2023), 102999. https://doi.org/10.1016/j.scico.2023.102999 20/03/2024.

[40] Amjed Tahir, Jens Dietrich, Steve Counsell, Sherlock Licorish, and Aiko Yamashita. 2020. A large scale study on how developers discuss code smells and anti-pattern in Stack Exchange sites. *Information and Software Technology* 125 (2020), 106333.

https://doi.org/10.1016/j.infsof.2020.106333 04/04/2024.

[41] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. 2010. The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In *2010 Asia Pacific Software Engineering Conference*. IEEE, Sydney, NSW, Australia, 336–345. https://doi.org/10.1109/APSEC.2010.46

[42] Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2009. Identification of Move Method Refactoring Opportunities. *IEEE Transactions on Software Engineering* 35, 3 (2009), 347–367. https://doi.org/10.1109/TSE.2009.1

[43] Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2011. Ranking Refactoring Suggestions Based on Historical Volatility. In *2011 15th European Conference on Software Maintenance and Reengineering* (15 ed.). IEEE, Oldenburg, Germany, 25–34. https://doi.org/10.1109/CSMR.2011.7

[44] E. van Emden and L. Moonen. 2002. Java quality assurance by detecting code smells. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.* IEEE, Richmond, VA, USA, 97–106. https://doi.org/10.1109/WCRE.2002.1173068

[45] Santiago Vidal, Hernan Vazquez, J. Andres Diaz-Pace, Claudia Marcos, Alessandro Garcia, and Willian Oizumi. 2015. JSpIRIT: a flexible tool for the analysis of code smells. In *2015 34th International Conference of the Chilean Computer Science Society (SCCC)*. IEEE, Santiago, Chile, 1–6. https://doi.org/10.1109/SCCC.2015.7416572

[46] Zhihong Wang, Hongru Ren, Renquan Lu, and Lirong Huang. 2022. Stacking Based LightGBM-CatBoost-RandomForest Algorithm and Its Application in Big Data Modeling. In *4th International Conference on Data-driven Optimization of Complex Systems (DOCS)*. IEEE, Chengdu, China, 1–6. https://doi.org/10.1109/DOCS55193.2022.9967714

[47] Morteza Zakeri-Nasrabadi, Saeed Parsa, Ehsan Esmaili, and Fabio Palomba. 2023. A Systematic Literature Review on the Code Smells Datasets and Validation Mechanisms. *ACM Comput. Surv.* 55, 13s, Article 298 (jul 2023), 48 pages. https://doi.org/10.1145/3596908 02/03/2024.